A HIERARCHICAL GRAPH MODEL OF CONCURRENT REAL-TIME SOFTWARE SYSTEMS

Paul David Stotts, Jr.
University of Virginia

Computer Science Report No. TR-86-12
May 23, 1986

A HIERARCHICAL GRAPH MODEL OF CONCURRENT REAL-TIME SOFTWARE SYSTEMS

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Paul David Stotts, Jr.

August, 1985

ABSTRACT

A model of a software system is introduced which is based on the theory of H-graph semantics. The model is intended to support a variety of performance studies for real-time concurrent programs. This work presents the formal concepts of the model, and discusses some analyses that can be applied to it.

The model of a concurrent system includes not only the application program but the host hardware as well, viewed as a collection of procedures with duration but no structure. Each procedure model in the system has three components: a data model, a static program model, and a control flow model. Concurrency is represented in the control flow component by a marked, timed Petri net which is distinguished by its notion of place duration and its restricted firing rule. A technique using Parallel Flow Graphs is introduced for producing well structured Petri nets with obvious interpretations in terms of software. With such models, we can automatically detect and correct conflicts in accesses to shared variables and data structures, as well as produce upper and lower bounds on execution times for portions of the modeled system. The analyses use a modified form of the Petri net reachability tree, one in which the effects of the concurrent firing rule and token timing are represented as restricting the set of reachable states.

ACKNOWLEDGEMENTS

I would like to express my gratitude to several people and institutions that have given me support and encouragement during the work reported in this document. First, my research has been financially supported in part by NASA Grant NAG-1-467, by U.S. Navy NSWC Grant N60921-81-K-A370 to the University of Virginia, and by National Science Foundation Grant NSF-DCR-8502881. In addition, special appreciation goes to my academic advisor, Terrence W. Pratt, for lending me his patience and insight over the past few years. I thank Joe Wilson, too, for sharing not only his office and his technical experience, but for the many leisure discussions, softball games, and other unusual diversions that helped fend off the exhaustion of mental work. Finally, to Nancy Jo, who has been too long the wife of a graduate student, I express my love and gratitude for her endurance and understanding through the many evenings at home alone.

TABLE OF CONTENTS

LIST OF DEFINITIONS

LIST OF THEOREMS AND LEMMAS

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF SYMBOLS

Symbol		*page*
Φ	universal set of h-graph nodes	30
Ξ	universal set of characters	30
Δ	set of all atoms formed over Ξ	30
#	atom denoting the null, or empty, string	30
a	an arbitrary atom	30
g	an arbitrary extended directed graph	30
M	set of all nodes in a graph	30
E	set of all edges in a graph	30
m'	initial node in a graph	30
Ω	set of all graphs over Φ and Δ	30
h	an arbitrary h-graph	31
G	set of all graphs in an h-graph	31
V	value function for an h-graph	31
r	root graph of an h-graph	31
$\mathbb{M}(h)$	set of all graph nodes in h-graph h	31
Γ	set of all h-graphs over Φ and Δ	31
Ψ	set of all node values for h-graphs in Γ	31
V^+	extended value function for an h-graph	32
$gsel$	syntax of an arbitrary node selector for a graph	33
$Gsel$	set of all graph selectors over Δ	33
$[\![gsel]\!]$	function denoted by graph selector *gsel*	33
$/$	selector for the initial node of a graph	33
$\bigcirc$	arc traversal function for a graph	33
$\perp$	*bottom*, denoting an undefined function value	33
s	syntax of an arbitrary h-graph selector	34
S	set of all h-graph selectors over *Gsel*	34
$[\![s\cdot]\!]$	function denoted by h-graph selector s	34
D	data component of an HG procedure model	35
H	an arbitrary set of data states (h-graphs) from Γ	35
h_0	initial data state of a data model	35
A	sequence of formal argument selectors in a data model	35
a_i	an arbitrary formal argument selector in A	35
R	sequence of formal result selectors in a data model	35
r_i	an arbitrary formal result selector in R	35
α	the assignment function for h-graphs	36
$\hat{f}$	an arbitrary procedure, a mapping of arguments to results	37
f	syntactic name for the procedure $\hat{f}$	37
F	set of all identifiers serving as procedure names	37
w	an arbitrary procedure call	38
A	list of actual argument selectors in a procedure call	38
a	an arbitrary actual argument selector from A	38
R	list of actual result selectors in a procedure call	38
r	an arbitrary actual result selector from R	38

b	an arbitrary basic block, a sequence of procedure calls	38
SP	static program component of an HG procedure model	39
NP	set of all named procedures in a static program model	39
B	set of all basic blocks in a static program model	39
ξ	an arbitrary execution sequence of basic block sets	39
N	Petri net structure	40
P	set of place nodes in a Petri net structure	40
p_i	an arbitrary place in P	40
T	set of transition nodes in a Petri net structure	40
t	an arbitrary transition in T	40
I	function defining arcs entering Petri net nodes	40
O	function defining arcs leaving Petri net nodes	40
δ	multiplicity function for Petri nets	41
μ	an arbitrary marking of a Petri net structure	41
μ^{ν}	number of tokens component of μ	41
μ^{τ}	token age component of μ	41
τ	timing function for the places in a Petri net structure	42
μ_0	initial marking for a Petri net structure	42
TP	a marked, timed Petri net	42
CF	control flow component of an HG procedure model	43
β	basic block selection function	43
σ	label selector function	43
ρ	path selection function	43
$\overline{\mu}$	reachability set of a timed Petri net	43
ζ	an arbitrary configuration of a control flow model	43
ζ_0	an initial configuration of a control flow model	43
P_a	the set of active places in a control flow model	44
P_c	the set of completely aged places in a control flow model	44
T_e	the set of enabled transitions in a control flow model	44
P_e	the set of enabling places in a control flow model	44
T_d	the set of data enabled transitions in a control flow model	44
P_d	the set of data enabling places in a control flow model	44
T_f	the set of firable transitions in a control flow model	44
P_I	the set of participating input places in a control flow model	44
P_O	the set of participating output places in a control flow model	44
B_x	the set of executable basic blocks in a control flow model	44
ζ_f	a final configuration of a control flow model	45
π	an arbitrary HG procedure model	49
SS	an HG software system model	50
F	set of primitive procedures in an HG software system model	50
Π	set of procedure models in an HG software system model	50
π_0	the main procedure in an HG software system model	50
PFG	abbreviation for Parallel Flow Graph	52
ϕ	an arbitrary parallel flow graph	53
W	set of procedure calls contained in a parallel flow graph	53
Υ	the distinguished PFG node value *join*	53
g	directed graph component of a parallel flow graph	53
V	value function of a parallel flow graph	53
$\hat{\tau}$	timing function of a parallel flow graph	53
$\boldsymbol{\eta}$	set of nodes in a parallel flow graph	53
η	an arbitrary PFG node from $\boldsymbol{\eta}$	53
η'	initial node of a parallel flow graph	53
E_{ϕ}	set of directed arcs in a parallel flow graph	53

e_{ϕ_i}	an arbitrary PFG arc in E_ϕ	53
α, β, γ	arbitrary atomic arc labels in Petri nets and PFGs	75
S_j^w	set of selectors in block j which are written	98
S_j^{wr}	set of selectors in block j which are written, then read	98
S_j^{rr}	set of selectors in block j which are read twice in succession	98
C_ϕ	set of potentially concurrent basic blocks from a procedure model	99
C_ϕ'	subset of C_ϕ, elements of which are in conflict	100
ω	an indefinitely large number of tokens in a Petri net place	104
MC_ϕ	concurrency matrix of potentially concurrent basic block pairs	113
R	the concurrent reachability tree of a Petri net	138
τ_F	timing for a set of procedures F	141
n_π	number of times procedure π is called in a basic block	142
$T_{\min}^p$	minimum total residence time of a token at place p in a Petri net	143
$T_{\max}^p$	maximum total residence time of a token at place p in a Petri net	143
$E_{\min}^p$	minimum token residence time at place p due to block execution	143
$E_{\max}^p$	maximum token residence time at place p due to block execution	143
$R_{\min}^p$	minimum token residence time at place p due to resource delay	143
$R_{\max}^p$	maximum token residence time at place p due to resource delay	143
$S_{\min}^p$	minimum token residence time at place p due to synchronization	143
$S_{\max}^p$	maximum token residence time at place p due to synchronization	143
$n_{\min}^p$	minimum number of tokens that may reside at a place p	144
$n_{\max}^p$	maximum number of tokens that may reside at a place p	144
$T_{\min}^{pipe}$	minimum token residence time in a normal pipeline	145
$T_{\max}^{pipe}$	maximum token residence time in a normal pipeline	145
$\alpha_{\min}^p$	minimum token delay at place p, obtained from net markings	147
$\alpha_{\max}^p$	maximum token delay at place p, obtained from net markings	147
$T_{\min}^{body}$	minimum token residence time in a loop body	149
$T_{\max}^{body}$	maximum token residence time in a loop body	149
lb, ub	lower, upper bound on number of executions of a loop body	149

Chapter 1

INTRODUCTION

A Hierarchical Graph model of a concurrent software system is a formal representation of interacting algorithms, amenable to analysis and transformation. The context of this work is the broad class of concurrency models for the procedural, von Neumann computational paradigm. In this chapter, a distinction between paradigms and models is first discussed briefly. The structure of the report is then outlined, and the major contributions of the research are summarized.

1.1. Computation paradigms and concurrency

Of the various computing paradigms that current programming languages support, the dominant one is still that of the von Neumann machine--the imperative, procedural style of programming. Originally offering only a sequential view of computation, this paradigm developed first, probably because sequential computation was a natural starting place for systems with a single hardware processing element. Functional separation of computing tasks followed, and then physical separation of computing devices. The natural outgrowth of this progression (or the cause in some cases) was the realization that some algorithms are not inherently limited to sequential performance of their individual operations. New computing paradigms were developed to provide insight into concurrent computation. In addition to extensions of the von Neumann paradigm to include multiple sequential machines, paradigms like data flow computation and functional programming are now being studied which have no inherent notion of necessarily sequential computation. Under these views, if several operations are to proceed in sequence, it is because of data dependencies rather than the semantics of the computation system.

The ability for independent operations to proceed in parallel is common to most currently used computing paradigms. The methods by which parallel activities are expressed, controlled, and understood are termed *models of concurrent computation*. Two models may be based on a common computation paradigm and yet be very different, depending on the aspects of computation that each emphasizes or ignores. A model necessarily abstracts away some of the complexity of the modeled system, in order to make more accessible to analysis the features of interest.

Many different models of parallel computation are currently under theoretical study or in practical use as programming languages. One of the difficulties in programming concurrent activities that solve problems is ensuring the correctness of the solution. The communication and synchronization patterns must allow the desired interaction among the concurrent activities and be shown to be free of deadlock and starvation; the algorithms must be shown to eventually terminate (or perhaps not to do so); a guarantee should be given that the program will produce some (desirable) results. Proving such properties is traditionally difficult for sequential algorithms; for concurrent ones, clear techniques have only lately begun to emerge [23,25,48]. Proving correctness is just one of several analyses that a programmer might wish to perform on concurrent programs. Others include determination of resource use, proving time constraint satisfaction, strong type checking across module boundaries, and determination of possible aliasing of variables.

This report presents our work on the modeling and analysis of concurrent computation in the context of software systems with real-time execution constraints. We have several motivations for this investigation. Programming environments have begun to evolve towards tool-oriented operation; a user produces a program with the aid of the environment and then uses analysis tools to guide verification and transformation of the product. Only recently has the emphasis been directed towards having a unified, mathematical model serve as the basis for the operation of the entire environment: development, analysis, transformation, and code generation for multiple targets. The Diana intermediate representation [1] for the Ada® language [2] is an example of this emphasis, but it is not mathematically rigorous enough to form the basis for an entire environment. We offer a computation model that provides this capability. Because code can be generated directly from the model, and because the analysis and transformation techniques presented are language independent, the source text from which a model is generated can be written in various languages.

A second goal is the inclusion of the host machine in the computation model of a software system. We feel that adequate timing analysis cannot be performed on software without regard for the hardware that is to execute it. A program that runs on two different hosts must be viewed as two different computations for real-time purposes. We present a model that allows the host architecture to be modeled as a collection of procedures, just as the software is. A system model can include all aspects of the execution environment: the machine; portions of an operating system; the language support routines; the application program.

1.2. Overview of topics

In Chapter 2 we examine several theoretical models of concurrent computation and discuss their utility as the foundation for modeling concurrent programs and languages. We also look at the models underlying several concurrent programming languages, and examine their features for communication and synchronization of parallel activity, their amenity to verification and other analyses, and their general operating principles.

Building on the results of this past research, we present in Chapter 3 the formal definitions of a hardware/software system modeling theory. Termed the HG model of concurrent software systems, it is based on Pratt's theory of sequential program modeling

®Ada is a registered trademark of the United States Government, Ada Joint Program Office (AJPO).

with Hierarchical Graphs [56]. The HG model presents a concurrency model based on timed Petri nets, an extension to regular Petri net theory which allows representation of execution duration for the modeled software. The execution rule for the model contains a notion of true event simultaneity. A set of concurrent operations contributes to a single data state transformation rather than an arbitrary sequence of individual transformations, allowing the equation of a state change with the passing of a unit of real-time. The HG model has a design permitting modular analysis of programs and verification of the interaction of concurrently executing computations.

Chapter 4 describes a model construction technique based on a graphical syntax. Termed *parallel flow graphs*, these algorithmic structures provide a disciplined method of creating HG system models that do not have the potential complexity and difficulty of interpretation of models created in an *ad hoc* fashion. Parallel flow graphs can be used directly as an algorithmic description language, or they can serve as semantic guides for translating program text in common languages into HG models. We demonstrate this generality and utility with examples of existing parallel language constructs expressed as parallel flow graphs.

Chapter 5 describes the detection and correction of conflicts in access to shared data structures by concurrent procedures in an HG model. Incorrect mutual exclusion (incorrect synchronization) in a computation can invalidate the data state. Though Petri nets can express the mutual exclusion of critical regions, the parallel flow graph syntax has no explicit construct to do so. Instead, we rely on automatic identification of critical regions in a modeled procedure where shared structures are being accessed in a way which, if done incorrectly, could compromise the integrity of the data state. The Petri net is then altered to provide mutual exclusion of these critical regions. The solution of this problem is possible by analysis of a special form of the Petri net reachability tree. The tree is constructed to reflect only the states achievable under the concurrent firing rule.

Chapter 6 discusses the timing aspects of the HG model. The notion of system timing *consistency* is introduced, and algorithms are presented which construct a pair of consistently timed models, one giving the minimum duration of a system's execution and the other the maximum. Any behavior the modeled system can exhibit must fall between these bounds, which provide the basis for timing constraint verification. An extension to the concurrent reachability tree is introduced which makes use of the timing figures as Petri net state information. Aspects of model deadlock detection from this reachability tree are addressed as well.

We draw our conclusions from this investigation in Chapter 7. The report ends with a discussion of several topics which, based on these results, look promising for future research.

1.3. Contributions of this research

Several aspects of this work are novel. The HG model offers a unique view of concurrent computation, in that the granularity of concurrency is variable and the data state has both distributed and shared aspects. A modeled system has a hierarchical structure, presenting each procedure as an independently analyzable object. The formalism provides a unified treatment for a software system and the hardware on which it executes, allowing various timing analyses to be performed on a model. Though others have used timed Petri nets to express control aspects of a computation, the concurrent transition firing rule applied to these nets is uncommon. It allows the equation of state sequences with time sequences and thereby provides a measurable notion of simultaneity. Though the timing analysis is partially automatable, only the beginning of a general solution is obtained in this work.

Parallel flow graphs not only provide a general syntax for expressing concurrent activities, but also play an important role as a vehicle for producing well-formed concurrent system models. Such models are automatically derivable from the graphical syntax, and the Petri nets produced from them are of a simpler class than the general nets. The graphical syntax is convenient for expressing algorithmic interactions, but can be cumbersome to use without appropriate tools and graphical output devices. Parallel flow graphs are general enough to express many major concurrency constructs succinctly, but they are not sufficient to model some interactions, notably the Ada *rendezvous* [2]. They also lack explicit nondeterminism in their present form.

The analysis techniques presented for detection and mutual exclusion of accesses to shared data structures are fully automatable. Moreover, the method for adding mutual exclusion to the model is independent of the methods whereby conflicts are identified. The analyses depend on the new form of reachability tree, which expresses in finite form the subset of normal Petri net states obtainable under the constrained firing rule, with token duration. Though the altered trees are adequate for solving the problems presented in this report, they offer no extra leverage over normal reachability trees in solving long-standing Petri net problems related to state reachability.

Chapter 2

MODELS OF COMPUTATION

As a context for the new computation model developed in Chapters 3-6, we review the features of several models of computation, divided into three categories: theoretical models, process-based language models, and non-process-based language models. In addition, some aspects of distributed computation and distributed models are discussed briefly. From the extensive amount of literature available covering these topics, only the more relevant works are mentioned here. A comprehensive overview of the field of parallel computation is offered by Filman and Friedman [22].

2.1. Theoretical models

Theoretical models of concurrent computation are developed to provide a mathematical basis for understanding the nature of computation. Usually they are not intended to be used directly as programming language models, as their generality provides more freedom than is necessary for writing algorithms.

2.1.1. Petri nets

The introduction of Petri nets by Petri in 1962 [52] marked one early attempt to mathematically formalize the concept of parallel computation. Complete presentations of these nets and their use are given by Miller [45] and Peterson [50,51]. The syntax is graphical, consisting of nodes of two types, with directed arcs connecting the nodes. A *place* node can hold *tokens*; it may serve as an *input* place or as an *output* place for a *transition* node. The state of a Petri net is represented by a *marking*, which is a mapping of each place in the net to a finite, possibly zero, number of tokens. A transition node becomes *enabled* when each of its input places contains one or more tokens. *Firing* a transition removes a token from each transition input place and deposits a token in each output place. If two or more transitions have common input places, the transitions are said to be *in conflict*. When two transitions in conflict are simultaneously enabled, the one to fire is chosen nondeterminately. If several arcs leave a transition, then a token is created for each outgoing arc when the transitions fires. Figure 2.1 shows a small Petri net with a sample marking of places 1, 2, and 4 having one token each, the others none. Transitions 2 and 3 are thereby enabled, with execution nondeterminately choosing one of them to fire to effect the next state change.

In theory, modeling of the concurrent activity in a computer system with Petri nets is a straightforward operation: the states of the system are represented by places and markings, while the actions are assigned appropriate transition nodes. Petri nets are also a natural model for data flow computation (discussed below); transition nodes are associ-

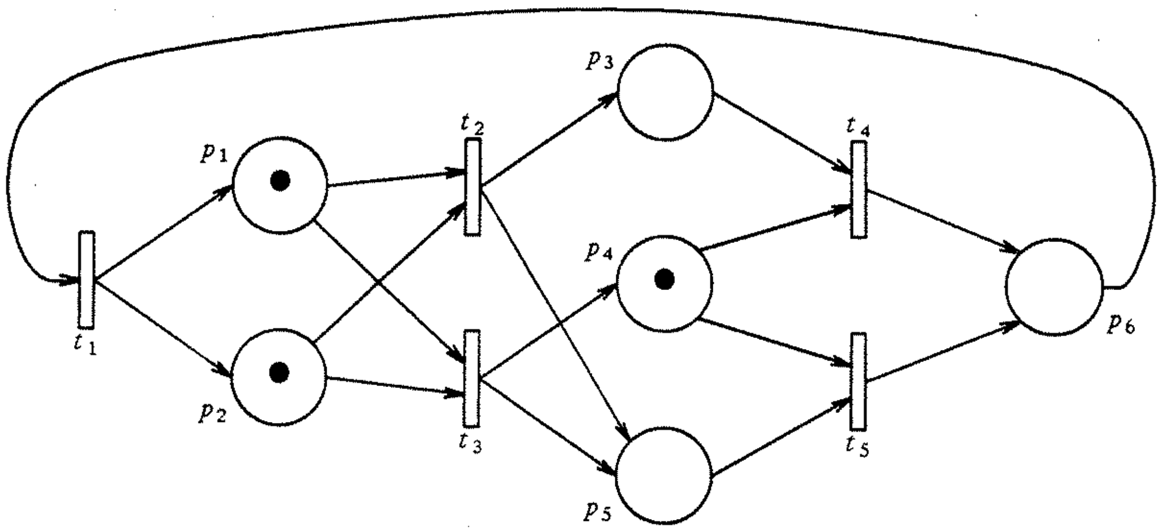


Figure 2.1 Petri net

ated with fine-grained actions such as the addition of two integers. The complexity of a net, however, grows exponentially with the number of its transition nodes. Practical use of this modeling technique alone, then, is limited to small systems or portions of systems, and programs of only moderate size. Examples of Petri net modeling of parallel activity can be found in [42, 61, 19, 46].

A common extension to basic Petri net theory is the inclusion of execution times on net events. The earliest work uses deterministic timing figures. Ramamoorthy and Ho [58] employ a model in which a duration is associated with each transition in the system, indicating how long the firing of the transition lasts. With these durations, they are able to compute for completely cyclic nets the number of times each transition will fire in each cycle of the net execution, and the minimum cycle time (maximum performance). Coolahan and Roussopoulos [15] present an alternative view in their model. A time is associated with each place, and a token is assumed to reside at a place for that number of state changes before it can participate in enabling any following transitions. From the timing figures is calculated the frequency with which each transition fires, relative to a base place/transition cycle called the *clock*. The average wait between enablings is also computable for each transition in the net.

The two forms of timing are functionally equivalent, though timing the transitions destroys the notion of instantaneous firings in the basic theory. Moreover, the notion of enforced duration for tokens at places is a more natural extension since, in a normal Petri net, tokens often reside at places for several state changes anyway. Merlin [44] presents a model that falls in between these two views in its behavior. Two timing figures are associated with each transition, giving a range of time in which a transition must fire after it becomes enabled. Time in Merlin's model is not deterministic, since a transition t may choose to fire at any point in its time range on each enabling.

2.1.2. Parallel program schemata

Parallel program schemata [34, 45] present another mathematical notion of parallel program. The theory combines the actions specified by a sequencing automaton with a formalized finite store, represented as a set of memory cells and their possible values. A parallel program schema S is defined by a triple (M, A, T) in which M is the set of *memory locations*, A is the set of *operations*, and T is the *control*. For each operation $a \in A$, there exists an associated *number of outcomes* $K(a)$ with a termination symbol a_i for each outcome $1 \leqslant i \leqslant K(a)$, a set of *domain locations* $D(a) \subseteq M$, and a set of *range locations* $R(a) \subseteq M$. The control is an automaton (Q, q_0, Σ, τ) in which Q is a *set of states*; q_0, the *initial state*, is an element of Q; Σ is the *alphabet*, the union of the *initiation symbols* $\Sigma_i = \{a \mid a \in A\}$ and the *termination symbols* $\Sigma_t = \{a_1, a_2, \cdots, a_{K(a)} \mid a \in A\}$; and τ is the *transition function*, a partial function mapping $Q \times \Sigma$ into Q, and totally defined on $Q \times \Sigma_t$. Concurrency enters in that, once initiated, an operation must not necessarily terminate before others are started. The control can be visualized as a flow graph with states at the nodes (providing history sensitivity to a computation) and with operation names as arc labels. Figure 2.2 illustrates such a graph; the set of operations A is $\{r, s, t\}$, with the outcome numbers being $K(r)=2$, $K(s)=1$, and $K(t)=2$. Operations r and s can proceed simultaneously, so the control allows their initiations and terminations in any order. Both must terminate, however, before t can proceed.

As defined, the schema has no meaning assigned to the computation it specifies; it is simply a collection of atomic actions that read and write memory cells, each action being initiated and terminated under the governance of the control. To supply semantics to a schema, an *interpretation* I is required. The interpretation provides functions that associate values with the memory elements of a schema. Formally, I is fully specified by a function C mapping each i in M into a set $C(i)$ of possible values, by an initial memory contents tuple c_0, and by two functions for each $a \in A$: F_a and G_a. The element c_0 is or subprograms). Because of the use of a solitary hardware processor, most of the processes are inactive at any time. Processes are swapped onto and off the processor according to elapsed execution time and current resource use or requirement. Interaction among processes is limited to requests for services from standard routines common to all, like *allocate memory*; effort is usually expended to prevent user processes from interacting (interfering) with each other. Queuing theory is often used to analyze the structure and performance of the concurrent activities.

Most parallel programming languages in common use are based on models of concurrent computation that present a set of macroscopic, cooperating, sequential processes. Computation is accomplished by multiple sequential code segments that go about their business as independently as possible, sending messages or signals back and forth as needed to keep pace with one another. Process-based language models are derived by application of operating system process ideas to the procedures in a traditional sequential programming language. As such they are a logical first step in the development of practical concurrent computation.

The process-based languages discussed in the following sections are representatives of the class of languages termed *imperative*, or *procedural*. They express computation as a collection of statements that prescribe some step-by-step sequence of events. Because of these sequential code segments, imperative programs contain constraints on action executions other than those necessitated by data dependencies (termed *precedence constraints*). The *granularity* of true parallelism is limited by the sizes of the sequential routines, which tend to be at least a dozen or so language statements. Smaller routine sizes can introduce inefficiency because of the mechanics of process invocation, similar to the effect of excessive procedure calling in a sequential program.

Process-based languages can be further classified according to methods of *communication*, for which two major schools of thought exist. One communication technique is based on the sharing of data areas, as used in Concurrent Pascal and Ada. Though several processes may have the rights to read and alter the same data, often the integrity of the data is protected with some mechanism that sequences accesses, disallowing simultaneous attempts to record new values. Another communication method is message passing, as used in Communicating Sequential Processes and Gypsy. Copies of data values are transported from one locale to another, or signals sent to indicate the existence of interesting data conditions.

Verification of programs in process languages is a two-step procedure. First, standard verification techniques are applied to the individual sequential routines, to prove their compliance with formal specifications. Then the network of communication and synchronization operations governing process interaction must be shown to be free of deadlock, starvation, race conditions, and other undesirable properties. Additionally, the network must be shown to guarantee the attainment of desirable program states, a property termed *liveness* by Owicki and Lamport [49].

2.2.1. Concurrent Pascal

Designed and implemented by Per Brinch Hansen in the early 1970's, Concurrent Pascal [28] operates on the principles of monitors, which were first developed by Brinch Hansen [27] and Hoare [30]. A fixed number of processes is declared, and all are invoked when a program execution is initiated. The process topology is static, and the processes communicate by altering shared data structures via calls to monitor procedures. Synchronization is provided *gratis* since the semantics of a monitor force mutual exclusion of processes on its data. Monitor procedures may place calling processes on queues for periods of time, and awaken them as well, providing rudimentary real-time capabilities to the language; other real-time features like interrupt handling and flexible process scheduling are absent.

Concurrent Pascal attempts to provide static checking of monitor calls to prevent deadlock of concurrent processes. To accomplish this several restrictions are imposed on the language. Before a procedure can be called it must be declared. Procedure definitions in monitors cannot be nested, and monitor procedures cannot call themselves recursively. Monitor calls and system access rights are thus hierarchically ordered; the resulting tree ensures deadlock freedom without run-time checking or restarting of processes.

2.2.2. Ada

The concurrent programming features of Ada [2] are generically termed *tasking*. An Ada task consists of a set of *entry* definitions and a body of executable statements. An entry declaration is similar to a procedure declaration; an entry call is made by a process outside the task, but the code for the entry is executed by the owning task. An *accept* statement in the task body defines a point at which a particular entry can be executed. An outstanding entry call is not honored until the task executes a corresponding accept statement in its body; conversely, a task that arrives at an accept statement for which no entry call has been made must wait.

The synchronization forced on the calling and accepting tasks at an entry is termed a *rendezvous*. The rendezvous facility can be thought of as defining points in a task execution at which work may be done on behalf of other tasks. The code of an entry executes in mutual exclusion, so interprocess communication may be accomplished via sharing of common data areas. Parameters may also be passed at an entry call, so the Ada concurrency mechanisms have some of the flavor of both monitors and messages.

2.2.3. Communicating Sequential Processes

An early process language model of concurrency is that of C. A. R. Hoare's Communicating Sequential Processes, or CSP [31]. CSP presents a view of parallel processing consisting of a set of macroscopic routines executing asynchronously and concurrently.

operating on local data and exchanging messages to communicate signals and intermediate results. Both sender and recipient must agree to rendezvous; hence the term *synchronized send* is used to describe the interaction between CSP processes. Messages are not buffered, so a program wishing to communicate with an unavailable partner synchronizes by waiting until the co-communicant arrives at the corresponding message operation in its own execution.

CSP offers little aid for verification of the possibly complex programs one can produce in its notation. It also has no real-time programming features. However, the model was presented as an incomplete experiment, and later efforts have resulted in extensions [62, 37] and axiom systems [5] aimed at verification of CSP programs.

2.2.4. Gypsy

Developed by Donald Good and others in the late 1970's, Gypsy [24] is designed for production of verifiable system software. Towards this end, the language is modular and allows inclusion of Hoare-like assertions on the state of program variables. The compiling environment depends heavily on a theorem prover for verifying the consistency of the assertions with the executable code. Global data is absent and aliasing is disallowed (by both static and run-time checks).

The concurrent computation model has procedure invocations as processes; communication between processes is provided solely by messages passed in variable-length (but finite) buffers. Synchronization is achieved via three mechanisms: the kernel provides mutual exclusion of processes on buffers when reading or writing a message; multiple process executions are started by a *cobegin* statement, the semantics of which prevent the caller from proceeding until all spawned processes terminate; and an *await* statement allows a process to explicitly pause until a certain data condition holds. Await can have several conditions that restart a process, thereby enabling nondeterminate computation in Gypsy. Forcing the routine invoking a cobegin to wait for termination creates a hierarchy of process invocations that guarantees freedom from some types of deadlock.

2.2.5. Modula and Modula-2

Defined by Niklaus Wirth in the middle 1970's, Modula [65] is an extension of Pascal to handle concurrent and real-time systems programming. Abstraction of data is supported by the extended scope rules of modules. For real-time programming, Modula provides direct communication between an executing program and physical devices in its environment, and facilities for fielding interrupts from these devices. Concurrency is achieved with parallel *processes*, assumed to be executing on a uniprocessor and scheduled according to a discipline built into the language definition. Processes communicate by exchanging messages via the *wait* and *signal* primitives.

Modula-2 [66], designed by Wirth in 1980, is intended to achieve the same goal as Modula, that of supporting real-time systems construction. The Modula process structure, considered too inflexible for low-level programming, is replaced in Modula-2 by a coroutine mechanism. Coroutines are created dynamically from procedure declarations. The user is then free to write modules that construct the concurrent process and communication mechanisms of choice. In keeping with this philosophy, no scheduling discipline is incorporated into the language. A coroutine is suspended when and only when it directly invokes another. Modula-2 is designed to be implemented on a uniprocessor, so many of the synchronization considerations in other concurrent languages do not apply.

The language facilitates the construction of large (multi-programmer) projects by allowing independently compilable code blocks and module specifications without corresponding bodies. For program verification, both Modula and Modula-2 provide only slightly more aid than Pascal. Meeting the timing constraints of real-time software requires the freedom of scheduling that Modula-2 provides. However, the potential concurrency--actual and simulated--presents verification problems more complex than those of sequential Pascal programs.

2.3. Non-process-based language models

Most languages having a computation model not based on processes are experimental; none has been developed commercially, though several are sufficiently advanced that they could be. The computation paradigms that gives rise to such models are significantly different from the von Neumann paradigm which spawned the process-based concurrency models; they have only recently been examined in detail. Non-process models have a concurrency unit of finer granularity than is normally associated with a procedure. Generally they cannot be implemented efficiently on existing computer architectures, so research is being conducted to discover machines to support non-process languages. We review several non-process-based computation paradigms: data flow, string reduction, and graph reduction.

2.3.1. Data flow

Some of the first work on the data flow computation paradigm is that of Dennis [20, 18]. Following the early work, a number of researchers developed appropriate languages and supporting machine architectures for this paradigm [21, 17, 6]. A comprehensive overview of data flow computation is given by Ackerman [4].

Programs in a data flow language are expressed as directed graphs with actions at the nodes. Data values propagate along arcs. Actions may execute when all required data values are present on the input arcs. Computed values are duplicated as required to place a copy on each output arc. The graph expresses only the precedence constraints imposed on action executions by data dependencies. Arbitrary sequencing encountered in standard imperative, process-oriented languages is not present, so parallelism is naturally expressed without explicit programmer notation. The granularity of concurrency depends on the actions at the graph nodes. Maximum parallelism is achieved by making the actions arithmetic operations, with individual operands on the arcs; thus, even expressions can be evaluated concurrently. Figure 2.4 shows a portion of a data flow program graph for

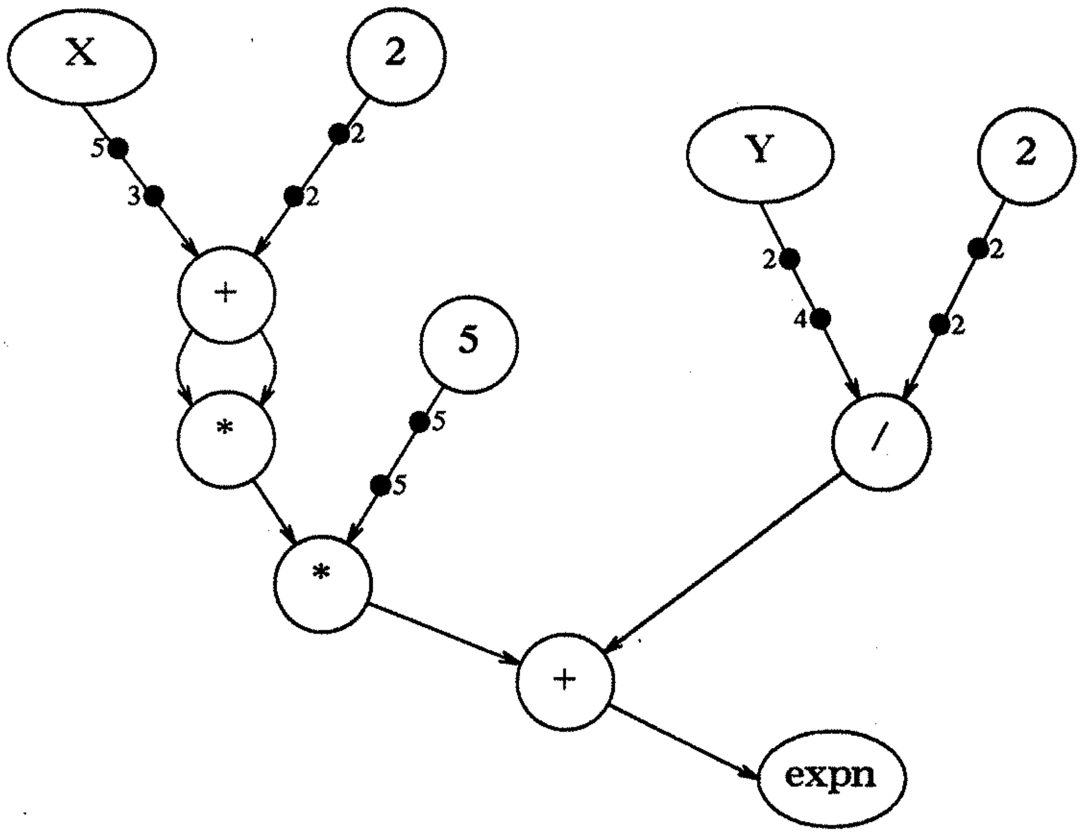


Figure 2.4 Data flow program graph

evaluating the expression $(x+2)^2 * 5 + (y/2)$; the computations in progress are for the (x,y) value pairs $(3,4)$ and $(5,2)$.

Since data flow computations have values as input and produce only values, no notion of variable is present in data flow languages. Consequently, no side effects can occur. Rather than directly writing programs as graphs, some data flow language designers choose a fairly standard, imperative syntax that a compiler then translates for execution. The languages VAL [3, 43] and ID [7] each offer such a syntax. The assignment statement then represents a *permanent binding* of a name to a value, not an alteration of a storage location. To aid in enforcing this view, data flow programs obey the *sin-*

gle assignment rule, which states that no name may appear on the left side of an assignment statement more than once across the scope of its binding. Creative interpretation is applied for looping—the scope is simply the loop body, and assignment (binding) is done at the start of each cycle.

Data flow programs are amenable to verification because of the single assignment rule. Ackerman [4] notes that for languages of this sort, called *definitional languages*, the assignment statements are like assertions on variable values; they may be used in verification of correctness.

2.3.2. String and graph reduction

Computation by reduction is perhaps better known as *functional*, or *applicative*, computing. The development of reduction languages and supporting architectures follows early work on the paradigm by Backus [8,9] and Berkling [11]. A review of current work in reduction and its comparison to data flow is presented by Treleaven *et al.* [63].

In the underlying computation paradigm a program is an expression, the application of a function to arguments (some of which may be function applications themselves). Control is expressed as recursion. Execution of a reduction language program consists of evaluation of the program expression. Unlike data flow, in which computation consists of sequences of fixed sized operations driven by the availability of operand data, reduction computation is carried out by successively rewriting, or *reducing*, the nested subexpressions into simpler form. A reduction terminates when all subexpressions evaluate to data values, which are irreducible.

As in data flow languages, reduction languages are definitional. No concept of variables or storage locations is included in the computation model. The assignment statement of imperative languages is replaced by the identifier definition, in which an expression is given a name by which it can be referenced. All subsequent references to a particular name use the one definition for it, a property known as *referential transparency*.

The functional notation imposes no spurious sequencing on a computation, expressing only the operation precedence created by data dependencies. This property, coupled with referential transparency, provides reduction languages with inherent parallelism and context independence for subexpression evaluation.

Reduction proceeds by replacing each identifier reference in an expression with its definition. Two methods of evaluation are commonly used: string reduction, and graph reduction. They differ in their handling of arguments and their preferred order of argument evaluation.

String reduction handles arguments with a by-value mechanism. When a subexpression is reduced, a copy of its definition is substituted into the computation, and reduction proceeds on each copy. Figure 2.5 demonstrates this reduction with a sample expression evaluation. Some inefficiency is introduced by duplicated effort, but addressing overhead in the hardware is low because the components of an instruction are directly referenced.

```
definitions

    m: 4     x: ( + m  1 )     a: ( * x  y )
    n: 3     y: ( * m  n )     b: ( - y  x )

evaluate  e: ( + a  b )

    ⟹  e: ( + ( * x  y ) ( - y  x ) )

    ⟹  e: ( + ( * ( + m  1 ) ( * m  n ) ) ( - ( * m  n ) ( + m  1 ) ) )

    ⟹  e: ( + ( * ( + 4  1 ) ( * 4  3 ) ) ( - ( * 4  3 ) ( + 4  1 ) ) )

    ⟹  e: ( + ( * 5  12 ) ( - 12  5 ) )

    ⟹  e: ( + 60  7 )

    ⟹  e: 67
```

Figure 2.5 String reduction

String reduction is especially suited to *innermost* (most deeply nested) function evaluation first, since it decreases the amount of copying that is done by only applying functions to fully reduced arguments. This is sometimes termed *data driven*, since data values are the innermost components of nested expressions. The architecture developed by Mago [40, 41] exemplifies the use of string reduction. The machine language he uses is Backus' FP [10], and reduction proceeds innermost first.

Graph reduction handles arguments with a by-reference mechanism. Each use of an identifier causes a pointer to its definition to be substituted into the computation, hence the idea of a graph. No duplication of effort results, but addressing overhead is greater since pointers must be traversed. Since structures are references, they can be manipulated without being reduced, making graph reduction well suited to *outermost* function evaluation first, as shown in Figure 2.6. A function can remain unevaluated until it is explicitly required in a computation. Then, once a subexpression has been reduced, further references to it receive the reduced value and are spared the work of reduction. The term *demand driven* is used to describe this property. AMPS [35] is a computation system based on graph reduction. The language used is a variant of LISP called FGL (Flow Graph LISP), described in [36].

2.4. Distributed models

Distributed computation might be considered as concurrent computation in which some assumptions about the underlying hardware can no longer be made without penalty. The extra considerations are concentrated in the areas of communication and data protection. In concurrent computation on single machines or on multiple machines that are in close proximity, communication of signals and other data among execution units is usually relatively fast and inexpensive. Transmission is also assumed to be reliable, so all messages arrive at their destination in a timely fashion, ungarbled. In a distributed system, in contrast, communication costs can be very high and the transmission medium

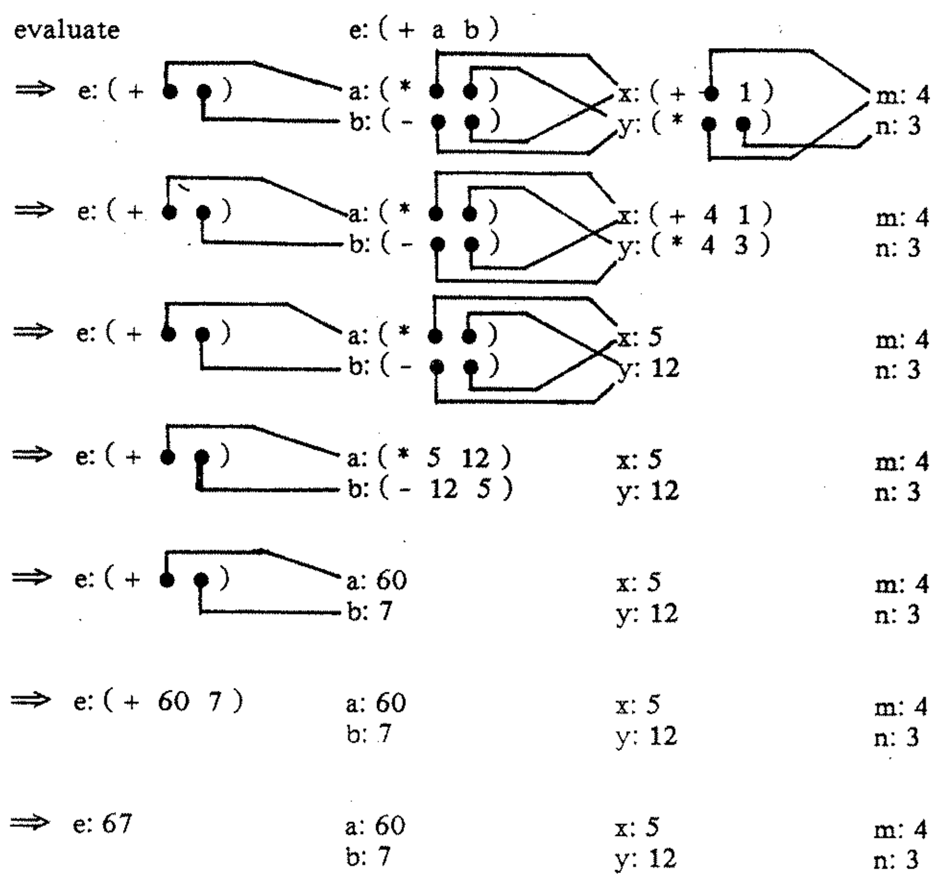


Figure 2.6 Graph reduction

itself unreliable. Consequently, the model cannot be simplified by assuming away the network costs of time and lost transmissions.

Research has been conducted to determine the essential differences between distributed computation and general concurrent computation. Liskov has defined some primitive operations required for distributed models [38,39]. Several languages have been designed for expressing computation on physically distributed pieces of hardware [14,29]. Operating systems have been constructed for controlling distributed processor systems [47,67,68]. This research project is not explicitly concerned with distributed computation; however, the proposed model does not preclude extension to distributed computation in later work.

2.5. Summary

Reviews are given of prior work in the field of concurrent computation modeling. In the class of theoretical models, we consider Petri nets, parallel program schemata, and process algebra. Features of each model are identified that are desirable in achieving the goals of our work. We then survey several concurrent programming languages which follow the procedural, von Neumann computation paradigm, notably Concurrent Pascal, Ada, CSP, Gypsy, Modula, and Modula-2. The underlying concurrency structure in each is examined, with the goal of discovering the features our formalism must possess in order to be suitable for modeling programs in such languages. Several non-process-based computation paradigms are then examined, including data flow computation, string reduction, and graph reduction. In the final section we look at the extra requirements that physical distribution of computing elements places on concurrency models.

Chapter 3

THE HG SOFTWARE SYSTEM MODEL

In this chapter we present the formal model of a software system for concurrent, real-time computation. The model is intended to provide the mathematical basis for various analyses and transformations of the modeled system, notably those described in the later chapters of this report. The first section places our work in the context of earlier research into Hierarchical Graphs and their applications. The following sections present the modeling formalism as a collection of component definitions, examples of use, and associated discussions of interpretation. Most notable are the development of the timed Petri net; its representation of software system attributes; the concurrent transition firing rule and its notion of simultaneity in real-time.

3.1. Overview of Hierarchical Graph modeling

Hierarchical graph structures, developed by Pratt [53], have been studied in various forms for the past fifteen years by several researchers. The earliest use was in operationally specifying the semantics of programming languages [54,55]. A program was expressed as a hierarchical graph that contained both data graphs and a code graph; the behavior of a standard automaton on executing the code graph defined the semantics of the instructions, and the state of the data on termination provided the results of computation.

Hierarchical Graphs (HG) now provides a complete, mathematically based programming language, the foundation of a software specification and analysis system [56]. Rather than defining the semantics of an entire language, the theory models individual programs. The text of a program guides the translation into the HG model of its static and dynamic semantics. This model has three parts: the data model, the static program model, and the control flow model. These components express the properties of the software that can be known prior to run-time. Such static information includes the composition and organization of the data to be manipulated, and the form and relationships among the instructions that describe the computation. The execution rule for the control flow component captures the dynamic nature of the software being modeled, in that it describes all possible execution sequences. In this sense, the control component is a generalization and a simplification of the automaton of the older theory. Features that may only be studied at the time of execution are available through this dynamic aspect of the model. A mathematical basis in set theory and formal grammars provides the framework in which programs are analyzed.

In Pratt's sequential formulation, the data model is composed of a set of states, each an h-graph. The possible data states are defined by a formal graph grammar. The static program model is a set of basic blocks, each a sequence of procedure calls. The control flow model is a deterministic finite automaton (DFA) which describes all potential sequences of basic blocks that execution of the model can produce. Given the current state of the system, and a value from the data state, the DFA indicates which block executes next, and the block dictates a certain transformation on the data state.

The concurrent computation model presented here differs from the sequential version in several respects. The data model has been retained, but in a slightly altered form; the graph grammar is absent, and some locations in each state are distinguished for argument passing. The static program model is basically unchanged, but in the control flow model a timed Petri net replaces the DFA for specifying possible block execution sequences. The Petri net enables expression of parallel control paths as multiple tokens in the net.

The HG software system model contains desirable features from several of the theoretical computation models presented in the previous chapter. We have chosen to separate the computation into distinct components, much as in parallel program schemata [34,45]. The data state corresponds roughly to the set of memory locations used in parallel program schemata, though the notion of structure in our model is distinct. The second component is the collection of instructions, grouped into sequential blocks according to the demands of the algorithm; a parallel program schema has no such grouping on its set of instructions *per se*. The third component is the control, which describes all possible sequences of instructions that an execution can produce. Whereas a parallel program schema employs a deterministic finite automaton as the control component, we choose a modified form of Petri net because of the interpretations we can give the places and transitions in relation to the instruction blocks. We add the notion of duration to the places in the Petri net to model the fact that real (as opposed to idealized) computations take time to complete. This quantifies somewhat the idea in parallel program schemata that initiation and termination of an operation are two different events.

Another advantage of choosing Petri nets as the basis of the control component of the HG software system model is that the model can be made hierarchical, like V. R. Pratt's process algebra [57]. In a Petri net, any place or transition in the network (indeed, any subnetwork) can be replaced by a more complex Petri net, thereby providing details of the computation that the simpler subnet represents.

Finally, the granularity of concurrency is variable in the HG model, depending on the wishes of the modeler. If the algorithm designer wishes only to examine the concurrency at the level of the procedure (as in Ada tasks or CSP processes) then he may leave all lower level operations such as arithmetic, memory allocation, etc., as primitive procedures--ones with no model structure other than a known duration and effect on the data state. If the desired analysis is to go deeper, then the operating system calls and the language support features can be included in the model as procedures with full data and control components. Their respective operations would eventually invoke the machine instructions (primitive procedures). The model components for the machine instructions could represent actual existing hardware, or they might represent a simulated novel architecture for executing a different computational paradigm such as dataflow.

3.2. H-graphs, selectors, and the Data Model

The definitions in the following sections present the basic tenets of the HG theory for modeling concurrent software systems. Definitions 1-1 through 1-4 and 1-9 through 1-14 are adapted from the theory of sequential software modeling introduced by Pratt. Definitions 1-5 through 1-8 are adapted from those presented by J. N. Wilson in his dissertation work on aliasing in h-graphs [64]. The remaining definitions represent the extensions we have developed to allow concurrent computation modeling.

This first collection of definitions introduces the formalism by which data is modeled. The theory presupposes two universal, finite base sets: the set Φ of *nodes*; and the set Ξ of *characters*. Given these, the following definitions lead to the concept of an h-graph, the basic model of data in this theory:

Definition 3-1: Atom
An *atom* is a finite sequence of characters from Ξ. The set of all atoms is denoted Δ, and $\Delta = \Xi^*$. The atom # denotes the null, or empty, string.

Definition 3-2: Graph
An *extended directed graph* (or simply *graph*) over Φ and Δ is a triple $g = \langle M, E, m' \rangle$ in which
$M = \{m_1, \cdots, m_k\}, k \geqslant 1,$ is a finite subset of Φ.
$E: M \times \Delta \rightarrow M$, a partial function
$m' \in M$
M is termed the *node set*, E the *arc set*, and m' the *initial node* of the graph g. If $E(m_i, a) = m_j$ then there is said to be an arc labeled a from node m_i to node m_j in g. For simplicity we assume that $m' = m_1$ and so write $g = \langle M, E \rangle$. The set of all graphs over Φ and Δ is denoted Ω.

Definition 3-3: H-graph

An *h-graph* over Φ and Δ is a triple, $h=\langle G,V,r\rangle$, in which

$G=\{g_1,\cdots,g_k\}$, $k\geqslant 1$, is a finite subset of Ω, such that each $g_i=\langle M_i,E_i\rangle$

$V\colon \bigcup_{i=1}^{k} M_i \rightarrow G\cup\Delta$

$r\in G$

G is termed the *graph set* of h; V is the *immediate value function*; r is the *root graph* of h. We assume that $r=g_1$ and write $h=\langle G,V\rangle$.

Related terms:

a. $\bigcup_{i=1}^{k} M_i$ is the *nodeset* of h, written $\overline{M}(h)$.

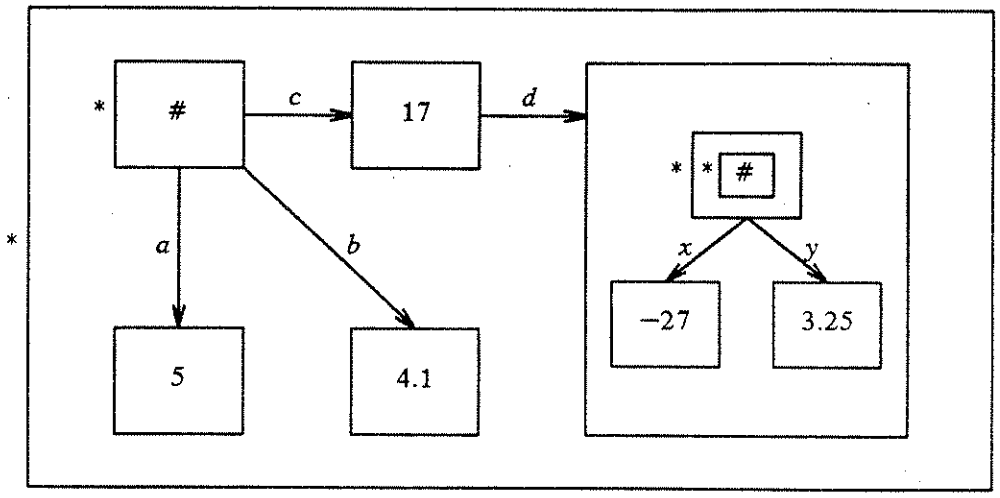
b. If $m\in\overline{M}(h)$, $V(m)$ is the *value* of m in h.

c. If $V(m)\in G$ then m is a *graph-valued node* of h; otherwise $V(m)\in\Delta$ and m is an *atom-valued node* of h.

d. The set of all h-graphs over Φ and Δ is denoted Γ.

e. The set $\Psi=\Gamma\cup\Delta$ is termed the set of *values*.

An h-graph is essentially a collection of directed graphs and atoms, and a function which maps the nodes in the graphs into these entities, thus creating a structural hierarchy



* = initial node of a graph

Figure 3.1 Example h-graph

among the graphs. Figure 3.1 illustrates these concepts. Often we find it convenient to use a string syntax for expressing the structure of h-graphs. In this syntax, identifiers are atoms, square brackets ([ ]) denote nodes, arrows with imbedded atoms denote labeled arcs, and vertically aligned arcs denote common parentage. The h-graph in Figure 3.1 is then expressed as:

```
[ [ # ]
    -a-> [ 5 ]
    -b-> [ 4.1 ]
    -c-> [ 17 ] -d-> [ [ [ # ] ]
                           -x-> [ -27 ]
                           -y-> [ 3.25 ]
                     ]
]
```

Definition 3-4: Extended value function

Given an h-graph $h=\langle G,V,r\rangle$, the *extended value function* for h is the function V^+: $\overline{M}(h) \rightarrow \Psi$. For $m \in \overline{M}(h)$, V^+ is defined by:

a. $V^+(m)=V(m)$ if $V(m)\in\Delta$;

b. $V^+(m)=h'$ if $V(m)$ is a graph; where h' is the h-graph $\langle G', V', V(m)\rangle$, with G' recursively defined as

   i. $V(m)\in G'$

   ii. If $g \in G'$ and m' a node in g such that $V(m')$ is a graph, then $V(m')\in G'$

   and with $V'=V \mid \overline{M}(G')$.

If $V^+(m)\in\Gamma$ then $V^+(m)$ is termed the *sub-h-graph defined by node m*.

The next four definitions are adapted from Wilson's work on aliasing detection in h-graphs [64]. A *graph selector* is used to identify a particular node in a graph, usually to obtain its value in an analysis. The definition is in two parts: the syntax, to describe the appearance of a graph selector, and the function denoted by the syntax, which describes the operation performed by a graph selector on a graph. The notation employed to distinguish these parts is [[*syntactic entity*]], where *syntactic entity* is the text of a selector and the [[ ]] brackets indicate the function represented by the contained selector.

Definition 3-5: Graph selectors

The syntax of a graph selector *gsel*, an element of the syntactic domain *Gsel* of all such selectors over Δ, is given by the production

$$
\begin{aligned}
gsel ::= & \\
& / \\
& \mid /a \\
& \mid /a_1 . a_2 \\
& \mid /a_1 . a_2 . a_3 \\
& \ldots
\end{aligned}
$$

where each $a_i \in \Delta$.

Definition 3-6: Function denoted by a graph selector

a. The *arc traversal function* $\bigcirc : \Omega \times (\Phi \cup \{\perp\}) \times \Delta \rightarrow (\Phi \cup \{\perp\})$ is defined, for $g = \langle M, E, m' \rangle \in \Omega$, $m_i, m_j \in M$, and $a \in \Delta$, to be:

$$\bigcirc(g, m_i, a) = \begin{cases} m_j & \textit{iff } E(m_i, a) = m_j \\ \perp & \text{otherwise} \end{cases}$$

$$\bigcirc(g, \perp, a) = \perp.$$

b. The function $[[gsel]]$ denoted by a graph selector $gsel \in Gsel$ is a function $[[gsel]] : \Omega \rightarrow \Phi \cup \{\perp\}$ which is defined by these cases:

Let $g = \langle M, E, m' \rangle \in \Omega$ and each $a_1, \cdots, a_n \in \Delta$.

- if $gsel \equiv /$ then $[[gsel]](g) = m'$; / is the *initial node selector*
- if $gsel \equiv /a_1$ then $[[gsel]](g) = \bigcirc(g, [[/]](g), a_1)$
- if $gsel \equiv /a_1 . \cdots . a_n$, $n > 1$, then

$$[[gsel]](g) = \bigcirc(g, [[/a_1 . \cdots . a_{n-1}]](g), a_n)$$

c. A node m of a graph g is termed *selectable* if there is a graph selector *gsel* such that $[[gsel]](g) = m$.

Note that a selector $gsel = /a_1 . a_2 \cdots a_n$ defines a directed path in a graph from the initial node to the node selected by it. Thus, a node is *selectable* if there exists some path to it from the initial node. Consider the graph g that is the value of the top level node in the h-graph of Figure 3.1; some sample selectors for this graph, and their respective function values, are shown below. Note that the value of each selector application $[[gsel]](g)$ is actually the *node* represented by the outer brackets in each example. Node values are shown inside the brackets for clarity:

```
gsel            [[gsel]](g)

/        →      [ # ]
/b       →      [ 4.1 ]
/c       →      [ 17 ]
/c.d     →      [ [ [ # ] ]
                      -x-> [ -27 ]
                      -y-> [ 3.25 ]
                 ]
```

Selection of a node from the hierarchical structure of an h-graph is performed by an *h-graph selector*, or simply *selector*. H-graph selectors are syntactically the concatenation of one or more graph selectors. Semantically, a node is selected by repeating for each graph selector this procedure: apply the graph selector to the target graph, obtaining a node; apply the value function of the target graph to the node, obtaining a new target graph. The selection is started by using the root graph of the h-graph as the first target graph.

Definition 3-7: H-graph selectors
The syntax of an h-graph selector s, an element of the syntactic domain S of all such selectors over $Gsel$, is given by the production

s ::=
$gsel$
$\mid gsel_1 gsel_2$
$\mid gsel_1 gsel_2 gsel_3$
. . .

where each $gsel_i \in Gsel$.

Definition 3-8: Function denoted by an h-graph selector
a. The function $[[s]]$ denoted by $s \in S$ is a function $[[s]]: \Gamma \rightarrow \Phi \cup \{\perp\}$ defined by these cases:
Let $h = \langle G, V, r \rangle \in \Gamma$ and each $gsel_1, \cdots, gsel_k \in Gsel$.
- if $s \equiv gsel$ then $[[s]](h) = [[gsel]](r)$
- if $s \equiv gsel_1 \cdots gsel_k,\ k > 1$ then

$$[[s]](h) = \begin{cases} [[gsel_k]](V([[gsel_1 \cdots gsel_{k-1}]](h))) & \\ \qquad \text{if } V([[gsel_1 \cdots gsel_{k-1}]](h)) \in \Omega \\ \perp \text{ otherwise} \end{cases}$$

b. A node m of an h-graph h is termed *selectable* if there is an h-graph selector s such that $[\![s]\!](h) = m$.

Considering the entire h-graph h in Figure 3.1, some sample h-graph selectors and their respective function values are shown below. As in the previous selector example, the value of each selector application $[\![s]\!](h)$ is the *node* designated by the outer brackets. Node values are indicated for clarity:

```
s            [[s]](h)

/       →    [ [ # ]
                 -a-> [ 5 ]
                 -b-> [ 4.1 ]
                 -c-> [ 17 ] -d-> [ [ [ # ] ]
                                          -x-> [ -27 ]
                                          -y-> [ 3.25 ]
                                  ]
             ]

//a       →    [ 5 ]
//c.d/x   →    [ -27 ]
//c.d//   →    [ # ]
```

Note that the single "/" selector denotes the top level node in the h-graph, and that the value in that node is the graph g we looked at in the previous graph selector example.

Definition 3-9: Data model
Let S be a set of h-graph selectors. A *data model* D over S is a four-tuple $D = \langle H, h_0, A, R \rangle$ in which
$H \subset \Gamma$ is a set of h-graphs,
$h_0 \in H$ is the *initial data state*,
$A = \langle a_1, \cdots, a_m \rangle$ is the sequence of *formal argument selectors*, each $a_i \in S$ such that $[\![a_i]\!](h_0) \neq \bot$, and
$R = \langle r_1, \cdots, r_n \rangle$ is the sequence of *formal result selectors*, each $r_i \in S$ such that $\forall h \in H \left| [\![r_i]\!](h) \neq \bot \right|$

The data model describes the structure of the local data area for a procedure. The initial data state h_0 describes the structure of the data when the associated procedure begins its execution. The selectors A designate locations in the initial data h-graph into which argument values will be copied from the invoking environment prior to execution. Each formal argument selector must be well-formed, in that each must designate a unique node in the initial h-graph. The formal result selectors R likewise designate unique nodes, but these nodes are further required to exist in all h-graphs in the data model. Execution of a procedure causes a sequence of transformations from one h-graph h to another h', where both $h,h' \in H$. The set R then designates locations in the final h-graph of such a sequence (if the sequence terminates) from which result values will be copied back to the invoking environment. The concepts involved in procedure execution are formally specified in subsequent definitions.

3.3. Data transformations and the Static Program Model

The following collection of definitions presents the formalism for altering a data state h-graph under software control. The notion of a *procedure* as a transformer of tuples of atoms is coupled with the *procedure call*, a specification of which portions of a data state to transform. The collected procedure calls in an algorithm form its *static program model*. Note that in the next definition, that of *assignment*, alterations to an h-graph (data state) are limited to replacing the existing value of a node by an atom; no assignment of graph values is defined.

Definition 3-10: Assignment
The *assignment function*
$\alpha: \Gamma \times \Phi \times \Delta \to \Gamma$
is defined as follows. Let $h = \langle G, V, r \rangle \in \Gamma$ be an h-graph, $m \in \overline{M}(h)$, and $\psi \in \Delta$ an atom. Then
$\alpha(h, m, \psi) = h' = \langle G, V', r \rangle \in \Gamma$
such that:
a. For $m' \in \overline{M}(h')$,
$$V'(m') = \begin{cases} \psi & \text{if } m' = m \\ V(m') & \text{if } m' \neq m \end{cases}$$
b. If $m \notin \overline{M}(h)$ then $\alpha(h, m, \psi)$ is undefined and is written as

$\alpha(h, m, \psi) = \perp$.

In an assignment $\alpha(h, m, \psi)$, h is termed the *local state*, m the *selected node*, and ψ the *value assigned*.

The assignment function is the means by which one alters the values of nodes in the component graphs of an h-graph. If we let the h-graph in Figure 3.1 be h, then the effect of the sequence of assignments

$$\alpha(h, [[//a]](h), 14) \rightarrow h'$$
$$\alpha(h', [[//c.d/y]](h'), 6.75) \rightarrow h''$$
$$\alpha(h'', [[//c.d//]](h''), true) \rightarrow h'''$$

is to produce the h-graph h''' represented by

```
[ [ # ]
    -a-> [ 14 ]
    -b-> [ 4.1 ]
    -c-> [ 17 ] -d-> [ [ [ true ] ]
                             -x-> [ -27 ]
                             -y-> [ 6.75 ]
                     ]
]
```

in which the graph structures are the same as in h, but some atomic node values have been altered. Since the bracket notation [[$\cdots$]] denoting the function for the text of a selector can be cumbersome, it is omitted from some of the remaining presentation. The notation $s(h)$ is used to designate a node in an h-graph when it is clear from the context that its meaning is the same as $[[s]](h)$.

Definition 3-11: Procedure

A *procedure* $\hat{f}$ is a function

$\hat{f} : \Delta^m \rightarrow \Delta^n, \ m \geqslant 0, n > 0$

mapping m-tuples of atoms into n-tuples of atoms. A *named procedure* is a pair $\langle f, \hat{f} \rangle$ in which f is the identifier naming the procedure $\hat{f}$. Let F be the set of all names of procedures.

Definition 3-12: Procedure call

a. Let S be a set of h-graph selectors. A *procedure call* over S and F is a triple $w=\langle f, A, R\rangle$ in which

  $f \in F$ is the *called procedure*;

  $A=\langle a_1, \ldots, a_m\rangle$, each $a_i \in S$, is the list of *actual argument selectors*;

  $R=\langle r_1, \ldots, r_n\rangle$, each $r_i \in S$, is the list of *actual result selectors*.

 A procedure call w is normally written $r_1, \ldots, r_n := f(a_1, \ldots, a_m)$.

b. Given a procedure call $w=\langle f, A, R\rangle$, a procedure $\hat{f}: \Delta^m \rightarrow \Delta^n$ corresponding to f, and an h-graph $h=\langle G, V\rangle \in \Gamma$, the *result of execution of w in state h* is $h' \in \Gamma$, defined as:

- evaluate the argument selectors A in h to get an m-tuple of *argument* atom values: $d_i = V(a_i(h))$, $1 \leqslant i \leqslant m$
- evaluate the called procedure on these values to get an n-tuple of *result* atom values: $\hat{f}(d_1, \cdots, d_m) = v_1, \cdots, v_n$
- apply the *assignment sequence* defined by w to get the *result state*:
  $\alpha(h_i, r_i(h_i), v_i) = h_{i+1}$ where $h = h_1$ and $h' = h_{n+1}$
- if any $r_i(h_i)=\perp$, or if any $d_i \notin \Delta$, then the result state $h'=\perp$.

Definition 3-13: Basic block

a. A *basic block* $b=\langle w_1, w_2, \cdots, w_k\rangle$ is a finite sequence of procedure calls, ordinarily written $w_1; w_2; \cdots ; w_k$.

b. Given basic block b and $h \in \Gamma$, the *result of execution* of b in state h, written $b(h)$, is the h-graph h' in which $h'=w_k(w_{k-1}(\cdots(w_1(h))\cdots)$. If one or more of the procedure calls w_i is undefined on its arguments then the result of execution of b is undefined and we write $b(h)=\perp$.

c. The *assignment sequence* defined by execution of b in state h is the concatenation of the individual assignment sequences defined by each of the procedure calls in b.

A basic block is the main unit of concurrent software in the model. Each modeled procedure is composed of perhaps many blocks, any of which may be simultaneously executing with others in the procedure. The component procedure calls in a block, however, are assumed to be required by the algorithm to execute in sequence. We do not seek with our analysis to identify situations in which a single block can be dissected into two or more concurrently executable sub-blocks. A sample basic block to calculate the quadratic formula is:

```
Block ROOTS

/t1 := negate ( /b )
/t2 := sqrt ( /discriminant )
/t3 := mult ( /2, /a )
/root1, /root2 := quadratic ( /t1, /t2, /t3 )
```

in which the procedure names are suggestive of their functions. Note that procedure calls can return multiple results.

Definition 3-14: Static program model

A *static program model* is a triple $SP=\langle S, NP, B\rangle$ in which

S is a set of selectors,

NP is a set of named procedures, and

$B=\{b_1, \ldots, b_n\}$ is a set of basic blocks, each containing only selector names from S, and called procedures f such that $\langle f, \hat{f}\rangle \in NP$.

Given a static program model SP and an h-graph $h \in \Gamma$, the following terms can be defined:

a. Let $B' \subset B$ be a set of basic blocks. A *concurrent execution* of B' in state h, written $B'(h)$, is the execution in h of a basic block composed of an arbitrary interleaving of the procedure calls comprising the blocks in B'. Formally, let $n=\sum_{i=1}^{k} length(b_i)$, for $b_i \in B'$ and $k=|B'|$. An *interleaving* of B' is a basic block $b'=\langle w_1, \cdots, w_n\rangle$ in which each $b_i \in B'$ appears as a distinct subsequence, that is, no two block subsequences share any w_j in b'. Then $B'(h)$ is defined to be $b'(h)$, the result of execution of b' in h.

b. An *execution sequence* is a (possibly infinite) sequence $\langle B_1, B_2, \cdots\rangle$ of sets of basic blocks, each $B_i \subset B$.

c. Let $\xi=\langle B_1, B_2, \cdots\rangle$ be an execution sequence. A *data state sequence* defined by ξ for initial state h is a sequence of h-graphs $\langle h_1, h_2, \cdots\rangle$ in which

$h_1 = h$, and

$h_{i+1} = B_i(h_i)$.

Each h_i is termed an *intermediate state* of the sequence, and $B_i(h_i)$ is a concurrent execution of B_i in that state. Each pair $\langle h_i, h_{i+1}\rangle$ is termed a *state transition*. If for some i, $B_i(h_i) = \perp$, and i is the smallest subscript for which this occurs, then the data state sequence is *undefined* after h_i and is written as $h_{i+1}=\perp$. If the data state sequence is defined for every intermediate state, then if ξ is finite and terminates at B_{t-1} the sequence of intermediate states is also finite, and it terminates at h_t; otherwise the intermediate state sequence is infinite. For a finite sequence, h_t is termed the *final data state*.

Note that there can be many distinct concurrent executions of a set of basic blocks, depending on how the interleaving is chosen. Chapter 7 discusses the implications of this feature for the determinacy of a computation.

3.4. Timed Petri nets and the Control Flow Model

The previous definitions provide the formalism for modeling the data and textual aspects of a software system. We now introduce the portions of the model that deal with the control aspects of a computation, that is, describing the various possible sequences of parallel block executions an algorithm will allow. We employ an extended form of Petri net to do this. Parts of the following definitions are adapted from basic Petri net theory as presented by Peterson [51]. Though we employ the standard net structure, we have made modifications to the surrounding Petri net theory to allow time bounds to be placed on the components of a modeled software system. The definition of a marking has been extended to facilitate the timing.

Definition 3-15: Petri net structure
A *Petri net structure* is a quadruple, $N=\langle P,T,I,O\rangle$ in which
$P = \{p_1, \ldots, p_n\}$ is a finite set of *places* with $n \geqslant 0$;
$T = \{t_1, \ldots, t_m\}$ is a finite set of *transitions* with $m \geqslant 0$, and $P \cap T = \varnothing$;
$I: T \rightarrow 2^P$ is the *input function*, a mapping from transitions to sets of places;
$O: T \rightarrow 2^P$ is the *output function*, also a mapping from transitions to sets of places.

The net structure is a bipartite graph in which $P \cup T$ is the set of nodes. We think of the functions I and O as defining the directed arcs connecting places to transitions and transitions to places. It is convenient to overload their definitions with these extensions:

$$I: P \rightarrow 2^T$$

$$O: P \rightarrow 2^T$$

The two formulations of the functions describe the same set of arcs, so that if

$$I(t_i) = \{p_j\} \quad \text{and} \quad O(t_i) = \{p_k\}$$

then

$$O(p_j) = \{t_i\} \quad \text{and} \quad I(p_k) = \{t_i\}.$$

We will find it useful at times to be able to treat the two functions as taking either places or transitions as their domain elements.

We define a *multiplicity* function $\delta: P \times 2^P \rightarrow \{0,1\}$ that, given a place and a set of places, returns the number of times the individual place occurs in the set. This number is either 0 or 1, indicating either absence or membership[1]. The set of places supplied as a domain element to δ is usually $I(t_i)$ (or $O(t_i)$) for some t_i, and so δ is usually interpreted as telling whether a particular place is an input (or an output) of a particular transition.

Definition 3-16: Marking

A *marking* μ of a Petri net structure $N=\langle P,T,I,O\rangle$ is a function

$$\mu: P \rightarrow \{0,1,\cdots\}\times\{\cdots,-1,0,1,\cdots\},$$

mapping each place in the net into an ordered pair of integers. The marking can also be described as an n-tuple of pairs, $\mu=\langle\mu_1, \ldots, \mu_n\rangle$ in which each $\mu_i \in \{0,1,\cdots\}\times\{\cdots,-1,0,1,\cdots\}$ and $n=|P|$. The two representations are related by $\mu(p_i)=\mu_i$, where $p_i \in P$.

Each ordered pair in a marking provides information about a particular place in the net. The first component of an ordered pair is the number of tokens residing in that place. The second component represents the age of the "oldest" token, that is, the token of the current resident group that was first to arrive. For a particular ordered pair μ_i, we will use the notation $\langle\mu_i^V, \mu_i^T\rangle$ to distinguish the individual components. The meanings of

[1]The function δ is a restriction of the multiplicity function # presented in Peterson's text, in which he uses multisets of places to describe the inputs and outputs for transitions. The use of multisets allows more than one arc to connect a place p_i to a transition t_j in either direction; the multiplicity then would be any integer $m \geqslant 0$. We have restricted the definitions of I and O to use sets of places, thereby allowing at most one arc between any place and transition in each direction. The two formulations of Petri nets are equivalent.

these concepts are more fully explained below.

Modeling real-time software requires analysis techniques for determining the time bounds on execution of portions of the system. Towards that end, timing figures are included in the Petri nets used for programs. As other researchers have done [15], we associate with each place in the net a number of time units; transitions are still considered to fire instantaneously as in pure Petri net theory.

Definition 3-17: Timing

A *timing* τ for a Petri net structure $N=\langle P,T,I,O\rangle$ is a function

$\tau: P \rightarrow \{1,2,\cdots\}$,

mapping each place in the net into one of the natural numbers. The timing can also be described as an n-tuple, $\tau=\langle\tau_1,\ldots,\tau_n\rangle$ in which $n=|P|$ and each $\tau_i \in \{1,2,\cdots\}$. The two representations are related by $\tau(p_i)=\tau_i$, where $p_i \in P$.

Definition 3-18: Marked timed Petri net

A *marked timed Petri net* is a triple $TP=\langle N,\mu_0,\tau\rangle$ in which

N is a Petri net structure,

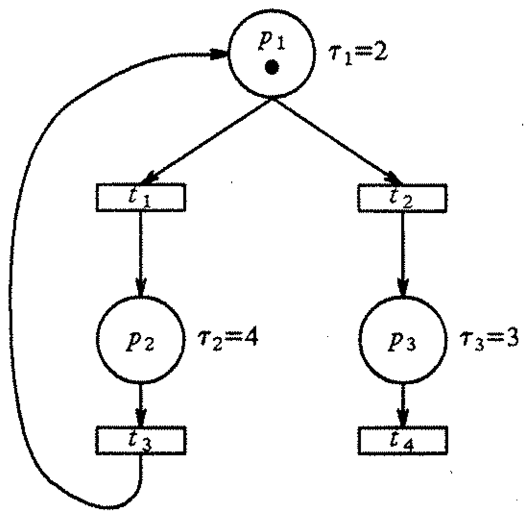
μ_0 is a marking of N, termed the *initial marking*, and

τ is a timing for N.

The marked timed net is sometimes written as a sextuple, $TP=\langle P,T,I,O,\mu_0,\tau\rangle$, combining the net structure, the initial marking, and the timing into one construction.

Figure 3.2 illustrates the concepts of Petri net structure, marking, and timing.

In a marked timed Petri net, a token arriving at a place p_i must reside in p_i for τ_i units of time before it can enable, or participate in enabling, the transitions that follow p_i. Thus, p_i could represent the condition that a corresponding instruction or group of instructions in the modeled program is executing, and the timing figure would then represent the relative amount of time required for it to complete. On completion (*i.e.*, after τ_i time units), the transitions following p_i are enabled by the resident token and may fire, signaling the start of execution of the code represented by the places that receive tokens thereby. If another token arrives at a place p_i in which a token is already resident, the new token does not begin to *age* until the previous token has resided for the



τ: $\langle 2,4,3\rangle$ μ_0: $\langle 1{:}2, 0{:}0, 0{:}0\rangle$

After 3 state changes (depending on whether t_1 or t_2 fires during the second state change) the marking could be

$\langle 0{:}0, 1{:}3, 0{:}0\rangle$ or $\langle 0{:}0, 0{:}0, 1{:}2\rangle$

Figure 3.2 Example of a marked timed Petri net

full duration τ_i and has caused a subsequent transition to fire.

Definition 3-19: Control Flow Model

Let $SP=\langle S, NP, B\rangle$ be a static program model. A *control flow model* over SP is a four-tuple $CF=\langle TP, \beta, \sigma, \rho\rangle$, in which

$TP=\langle P, T, I, O, \mu_0, \tau\rangle$ is a marked timed Petri net.

β: $P \rightarrow B$ is the basic block selection function,

σ: $P \rightarrow S$ is the label selector function,

ρ: $P\times\Delta \rightarrow 2^T$ is the path selection function, a partial function subject to these restrictions:

for $p \in P$ and $a_i, a_j \in \Delta$,

- either $\rho(p,a_i) \subset O(p)$ or $\rho(p,a_i)$ is undefined
- $\rho(p,a_i) \cap \rho(p,a_j) = \emptyset$, $a_i \neq a_j$

a. Let $\overline{\mu}$ be the set of all possible markings of the net structure in TP. A *configuration* of CF is a pair $\zeta=\langle \mu, h\rangle$ in which $\mu \in \overline{\mu}$ is termed the *current control state*, and $h \in \Gamma$ is termed the *current data state*.

b. An *initial configuration* of CF is a configuration $\zeta_0=\langle \mu_0, h\rangle$, where $h \in \Gamma$.

c. Given TP and a configuration $\zeta = \langle \mu, h \rangle$, where $h = \langle G_h, V_h \rangle$,

i. The set $P_a = \{p_i \mid p_i \in P \text{ and } \mu_i^\nu > 0\}$ is termed the set of *active places*.

ii. The set $P_c = \{p_i \mid p_i \in P_a \text{ and } \mu_i^\tau \leqslant 1\}$ is termed the set of *completely aged places*.

iii. The set of *enabled transitions*, that is, the set of transitions that the places in P_c fully enable, is defined as $T_e = \{t \mid I(t) \subset P_c\}$.

iv. The set of *enabling places*, that is, the places which are inputs to the enabled transitions, is defined as $P_e = \bigcup_{t \in T_e} I(t)$.

v. The set of *data enabled transitions*, that is, the transitions that are enabled and designated by the path selection function as eligible to fire, is defined as $T_d = \{t \in T_e \mid \forall p \in I(t), t \in \rho(p, V_h([\![\sigma(p)]\!](h)))\}$.

vi. The set of *data enabling places*, that is, the places which are inputs to data enabled transitions, is defined as $P_d = \bigcup_{t \in T_d} I(t)$. Note that $P_d \subset P_e \subset P_c \subset P_a \subset P$.

vii. The set T_f of *firable transitions*, that is, transitions that are data enabled and chosen to actually fire, is iteratively defined. Given a set of data enabled transitions $T_d{}^i$, the set $P_d{}^i$ of all input places to these transitions, and a set $T_f{}^i$ of transitions, then:

1. choose some $t \in T_d{}^i$
2. $T_f{}^{i+1} = T_f{}^i \cup \{t\}$
3. $P_d{}^{i'} = P_d{}^i \dot{-} I(t)$
4. $T_d{}^{i+1} = \{t \in T_d{}^i \mid I(t) \subset P_d{}^{i'}\}$
5. $P_d{}^{i+1} = \bigcup_{t \in T_d{}^{i+1}} I(t)$

The iteration begins by letting $T_f{}^0 = \emptyset$, $P_d{}^0 = P_d$, and $T_d{}^0 = T_d$. It terminates when $T_d{}^{i+1} = \emptyset$, at which point we define $T_f = T_f{}^{i+1}$. Note that $T_f \subset T_d \subset T_e \subset T$.

viii. The set of *participating input places*, that is, the places that actually lose a token due to the firing of a transition, is defined to be $P_I = \bigcup_{t \in T_f} I(t)$. Note that $P_I \subset P_d$.

ix. Similarly, the set of *participating output places*, that is, the places that actually gain a token due to the firing of a transition, is defined to be $P_O = \bigcup_{t \in T_f} O(t)$. Note that P_I and P_O are not necessarily disjoint.

viii. The set B_x of *executable basic blocks* is defined to be

$$B_x = \{\beta(p) \mid p \in P_I\},$$

i.e., the blocks associated with the places that each lose a token in the current state change.

d. Given a configuration $\zeta = \langle \mu, h \rangle$, a *next configuration* defined by CF is $\zeta' = \langle \mu', h' \rangle$ in which

i. μ' is defined to be

$$\mu_i'^\nu = \mu_i^\nu - \delta(p_i, P_I) + \sum_{t_j \in T_f} \delta(p_i, O(t_j))$$

$$\mu'^{\tau}_i = \begin{cases} \mu^{\tau}_i & \text{if } p_i \notin P_a \text{ and } p_i \notin P_O \\ \mu^{\tau}_i - 1 & \text{if } p_i \in P_a \text{ and } p_i \notin P_I \\ 0 & \text{if } p_i \in P_I \text{ and } \mu'^{\nu}_i = 0 \\ \tau_i & \text{if } p_i \in P_I \text{ and } \mu'^{\nu}_i > 0, \text{ or if } p_i \notin P_a \text{ and } p_i \in P_O \end{cases}$$

ii. h' is defined to be the h-graph resulting from a *concurrent execution* of B_x in state h, written $B_x(h)=h'$.

e. The pair of configurations $\langle \zeta, \zeta' \rangle$ is termed a *state change* for CF. A *computation of CF* is a (possibly infinite) sequence of state changes $\langle \zeta_0, \zeta_1 \rangle, \langle \zeta_1, \zeta_2 \rangle, \langle \zeta_2, \zeta_3 \rangle, \cdots$ starting from an initial configuration, which is more succinctly denoted $\zeta_0, \zeta_1, \zeta_2, \cdots$

f. A *final configuration* $\zeta_f = \langle \mu_f, h_f \rangle$ is a configuration in a computation for which $P_a = \emptyset$. If a configuration is reached in which $P_a \neq \emptyset$ and $\mu^{\tau}_i \leqslant 0$ for each $p_i \in P_a$, then the configuration is said to be a *deadlock*; the ζ_f is then defined as the first such deadlocked configuration. A computation which contains a final configuration is said to *terminate*.

The path selection function ρ serves as an oracle which, when a place has several transitions in its output set, specifies a particular subset of those transitions. From this set, one transition is chosen to fire, thus contributing to the state change. The set is selected based on a value from the data state h-graph, the location of which is designated by the selector produced by the label selector function σ. Using the oracle to select some paths over others during execution of a model does not alter the structure of the Petri net. We simply do not employ a completely nondeterministic choice at these branch points. The model does provide in the net structure all possible control paths. We can still perform analyses to determine what control states are permitted in a system without considering the makeup of the basic blocks themselves. When a particular set of basic blocks is considered, that is, when particular procedure calls and their effects on the data state are viewed, some paths may actually be excluded during the execution of a model with a particular control structure.

The set T_f of transitions that actually fire is some subset of T_d, all the ones that could possibly fire and are designated by the data state as eligible. For the analysis in the remainder of this report we assume that the size $|T_f|$ of this set is as great as possible at each state change, that is, we fire as many transitions per state change as the data constraints will allow. No transition that is enabled and selected by the data state as eligible will have its firing arbitrarily delayed until another state change, unless it is in conflict (as defined in section 2.1.1) with another eligible transition. This assumption creates a firing rule in which all transitions that can possibly fire are required to do so within the same state change in which they become eligible. The physical situation we wish to reflect with this firing rule is one in which enough hardware is available that no ready process must wait for a processor when executing the software system. Thus, all waiting in our model will be a direct consequence of the synchronization required by the algorithm rather than possibly an artifact of the host machine. As an extension, the added generality of allowing an arbitrary subset of the enabled transitions to fire would favor modeling systems in which the hardware does impose some waiting and other scheduling constraints on the execution. The final chapter on future research issues discusses the implications of selecting some subset of firable transitions which is smaller than the maximum.

Note that the opposite firing restriction to the one we have adopted is that the cardinality $|T_f|$ be always one. This generates the firing rule of normal Petri nets: from the set of all enabled transitions, one is chosen to fire and a new state results from the firing. The hardware-oriented interpretation that we are giving to the cardinality of the set T_f suggests that Petri nets with the normal firing rule are useful for real-time concurrency analysis primarily with a time-shared uniprocessor implementation.

Consider the iterative definition of the set T_f. The set is constructed so that as many of the enabled transitions as possible will fire, according to the values of the current data state. First we consider all the transitions which are enabled by fully aged tokens and have been designated by the path selection function as eligible to fire. One of these is selected and added to the firable transition set. The input places for the chosen transition are removed from the set of enabling places, and a new set of of data enabled transitions is found. This process is repeated until no data enabled transitions remain to be considered.

In constructing the next configuration from the current one, two decisions are made which the model does not constrain. One such decision comes in creating the set T_f from the set of data enabled transitions. The choice is made by selecting data enabled transitions one at a time, and recording which previously enabled transitions are disabled by the choice, until none remain. Since the block selection function β maps each place into a basic block which we consider to have been executing while a token in the place ages, the decision is thereby made to terminate the set of basic blocks associated with the input places for the chosen transitions, and to begin execution of the blocks associated with the output places. In each iteration, then, the choice of which single transition in $T_d{}^i$ to consider next encapsulates a notion of scheduling. The choice technique is left unspecified to allow different scheduling strategies to be modeled and explored.

The second unconstrained decision is made in determining the data state resulting from concurrent execution of the set of blocks B_x. An interleaving of the blocks in B_x must be created, and the model does not prescribe the method for its generation. Again the choice encapsulates a notion of scheduling, but on a somewhat lower level than the first decision. Selecting the interleaving involves putting some procedure call executions before others, and different interleavings can lead to different data states.

The notion of time units discussed in the definition of a timed Petri net has been equated to state changes in the control model: one state change represents the passing of one time unit. A timing τ_i on a place p_i, then, really means that a token arriving at p_i must reside there for τ_i control state changes before the following Petri net transition is enabled. We can think of each Petri net place p_i with timing $\tau_i > 1$ as being equivalent to a linear subnet of places and transitions, as shown in Figure 3.3. The subnet consists of

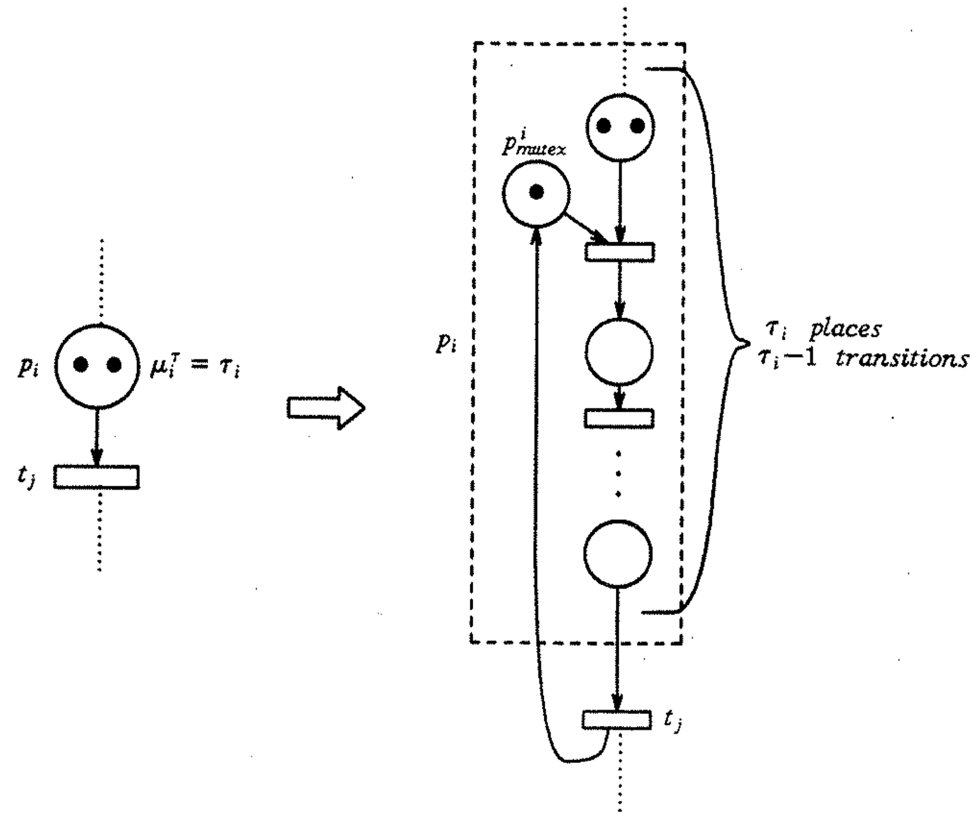


Figure 3.3 Equivalence of timed and normal Petri nets

τ_i places and $\tau_i - 1$ transitions, connected in the order $p^i_1, t^i_1, p^i_2, t^i_2, \cdots, p^i_{\tau_i - 1}, t^i_{\tau_i - 1}, p^i_{\tau_i}$. The transitions that are inputs to the original p_i are, in the replacement subnet, inputs to p^i_1; the transitions that are outputs of p_i become the outputs of $p^i_{\tau_i}$. A mutual exclusion place p^i_{mutex} is also required, since only one token at a time is allowed to age at each place in a timed net. We make p^i_{mutex} an input to the first transition t^i_1 in the sequence, and we make it an output of any following transitions t_j.

The original basic block $\beta(p_i)$ is associated with the final place p^i_τ in the expanded net. The other newly created places p^i_k, $1 \leqslant k < \tau_i$, all have null basic blocks associated with them, as does the mutual exclusion place p^i_{mutex}.

Applying the same execution rule to the expanded nets as we do to the timed nets of the control model--all transitions that are enabled must fire in the next state change--we see that the timed nets and the untimed, expanded nets are equivalent. Essentially, the expanded, untimed nets can be viewed as timed nets in which all places have a duration of one state change. Since the structure of the linear sequence of place/transition pairs in the untimed nets (representing a single place in the timed nets) precludes any waiting past the single state change, they all fire in sequence, taking exactly τ_i state changes to do so.

3.5. Procedure models and the Software System Model

The definitions in this final collection unify the concepts presented in the preceding sections. A *procedure model* and the function it computes are first defined as a named entity having a data state, some prescribed data transformations, and a control description. A collection of such models is then termed a *software system.*

Definition 3-20: Procedure model

A *procedure model* π is a four-tuple $\pi=\langle f, D, SP, CF\rangle$ in which

f is the procedure name,

D is a data model,

$SP = \langle S, NP, B\rangle$ is a static program model, and

$CF = \langle TP, \beta, \sigma, \rho\rangle$ is a control flow model over SP.

When invoked, a procedure model is executed by applying the transition rule of its control flow model to an initial configuration $\langle\mu_0, h_0\rangle$ where h_0 is an h-graph in D and μ_0 is the initial marking of TP.

Definition 3-21: Procedure represented by a procedure model

Given a procedure model $\pi=\langle f, D, SP, CF\rangle$, where $D=\langle H, h_0, A, R\rangle$, $A=\langle a_1, \cdots, a_m\rangle$, and $R=\langle r_1, \cdots, r_n\rangle$, the *procedure represented by* π is the function $\hat{f}: \Delta^m \rightarrow \Delta^n$, defined as follows:

a. Let $\langle d_1, \cdots, d_m\rangle \in \Delta^m$. The *argument transmission* sequence is

$$\alpha(h_i, a_{i+1}(h_i), d_{i+1}) = h_{i+1}, \text{ where } 0 \leqslant i \leqslant m-1.$$

To ensure that each $h_i \neq \perp$ we require that no a_i is a prefix of a_j, $i \neq j$. See definition 5-1 for the formal meaning of a selector prefix.

b. The *initial configuration* of CF is $\zeta_0=\langle \mu_0, h_m \rangle$.
c. Execute π as defined by the control flow model CF with initial configuration ζ_0 to get a final configuration $\zeta_f=\langle \mu_f, h_f \rangle$, if one exists.
d. If ζ_f does not exist, then $\hat{f}(d_1, \cdots, d_m)$ is undefined. If ζ_f does exist, then $\hat{f}(d_1, \cdots, d_m)=\langle v_1, \cdots, v_n \rangle$ where each $v_i=V(r_i(h_f))$ for $h_f=\langle G, V \rangle$.

For the purposes of this analysis, we require CF to be deterministic so that the result values of $\hat{f}$ are uniquely defined, *i.e.*, the procedure represented by each procedure model is a function. Note that if both scheduling decisions in the control flow model (choosing a transition and choosing an interleaving) are made using deterministic techniques, then this restriction will hold. If either choice is made nondeterministically, then the procedure represented by the procedure model is a relation rather than a function. The final chapter on future research discusses the implications of allowing nondeterministic control flow models.

Definition 3-22: Software system model
a. A *software system model* is a triple $SS=\langle \hat{F}, \Pi, \pi_0 \rangle$ in which
$\hat{F}$ is a set of named procedures, termed the *primitive procedures*;
Π is a set of procedure models, subject to the restriction below;
$\pi_0 \in \Pi$ is the *main procedure*.
The set of procedure models Π must be constructed so that if $\pi=\langle f, D, SP, CF \rangle \in \Pi$, where $SP=\langle S, NP, B \rangle$, then $\forall f' \in NP$ either $f' \in \hat{F}$ or $\exists\, \pi'=\langle f', D', SP', CF' \rangle \in \Pi$.
b. The *procedure represented by SS* is the procedure represented by the main procedure π_0.

A software system model defines a collection of procedure models and primitive procedures that describe all the computation in the software to be analyzed. Each procedure model is composed of basic blocks which call only the primitive named procedures or procedures which have a model in the system. The intent is to have the primitive procedures be the machine instructions of the host architecture. No structure is known for the computation each performs, but the effect each has on the data state and the time each requires is known. The procedure models are then the user-defined software components of the system.

3.6. Summary

The formal HG model of a concurrent software system is introduced and is seen to have some desirable features found in previous computation models: hierarchical decomposition into independently analyzable procedures; separate components for the different computation aspects of data, program text, and control flow; time (computation duration) as a measurable entity in the model; a notion of simultaneity in the execution rule for the model. The flow of parallel control paths in a modeled algorithm is represented by a Petri net, augmented by times on the places and by an execution rule allowing multiple transition firings to create a single state change. We present an interpretation of this model in terms of the parts of a software system and their execution on a parallel host architecture.

Chapter 4

CONTROLLING HG MODEL COMPLEXITY

The control flow model, as presented, is largely a general Petri net with some additions that enhance its suitability for real-time analysis. We find it convenient to introduce a technique for controlling the acceptable structure of these nets, that is, limiting the software modeler to using only a subset of the general Petri nets. The restrictions serve the same purpose in our theory that structured programming does for the generation of manageable algorithms--they limit achievable complexity but not expressive power. We describe in the first section the formalism of a notation for expressing concurrent algorithms, termed *parallel flow graphs*. In the second section we describe how to translate a parallel flow graph into its HG model components for analysis. As a demonstration of their utility for modeling software, the final section presents examples of concurrent control structures from existing programming languages expressed in the notation of parallel flow graphs.

4.1. Parallel flow graphs

To accomplish the goal of modeling concurrent computation with a Petri net structure of limited complexity, we view the static program model and the control flow model as a unified entity, represented in a graphical notation termed a *parallel flow graph* (hereafter abbreviated *PFG*). A program can be constructed as a PFG and then dissected into the two component models for analysis. The components of a Petri net produced from a PFG are easily associated with portions of the modeled software, thus ensuring that analysis is attempted only for HG models with reasonable interpretations.

Definition 4-1: Parallel flow graph

Let W be a set of procedure calls, S be a set of selectors, and Υ be a distinguished node value. A *parallel flow graph* ϕ over W, S, and Υ is a tuple $\phi = \langle g, V, \hat{\tau} \rangle$ in which

a. $g = \langle \eta, E_\phi, \eta' \rangle$ where

   η is a finite set of nodes,

   $\eta' \in \eta$ is the *initial node*, and

   E_ϕ is a finite set of arcs, each $e_{\phi_i} \in E_\phi$ of the form $\langle \eta_j, \eta_k, a \rangle$ with $\eta_j, \eta_k \in \eta$ and $a \in \Delta$, indicating that an arc labeled with atom a exists from node η_j to node η_k; the arcs in E_ϕ are subject to the restrictions stated below.

b. $V: \eta \rightarrow S \cup W \cup \{\Upsilon\}$ is a function mapping each node in g into either a selector, a procedure call, or the distinguished value Υ.

c. $\hat{\tau}: \eta \rightarrow \{1, 2, \cdots\}$ is a function that associates a positive, integral execution time with each node in the PFG.

Each node of a parallel flow graph contains either a procedure call to affect the data state, a selector to direct branching (and concurrency creation), or the distinguished value Υ (termed *join*) to perform synchronization (and concurrency deletion). Because each procedure begins with a single control path, the initial node η' may not contain Υ. The arcs between nodes represent the flow of control from one action in an algorithm to the next, and each arc has an atomic label associated with it. To ensure connectivity, each node in the graph must be on a directed path from the initial node. Obviously at least one arc, then, must enter each node (other than the initial node), but we place no upper limit on this number. The initial node may have no arcs entering it. Arcs leaving a node are governed by several constraints. A node containing a procedure call or a join may have no arcs leaving it, or it may have a single arc leaving it with the label on that arc being *null*, written #[1]. A selector node, or *branch node*, may have any positive number of arcs leaving it. The label on each of these arcs may be any atom from Δ, and they need not be unique, *i.e.*, an atom may serve as label for two or more of the out-arcs of a

[1]By default, an arc with no written label has the *null* label.

branch node. Figure 4.1 illustrates this synthesis with a portion of a PFG in which each s_i represents a selector and each w_i represents a procedure call. In subsequent figures involving PFGs we employ a notational shorthand for sequences of procedure call nodes. Rather than explicitly picture each node in such a sequence, we represent the entire

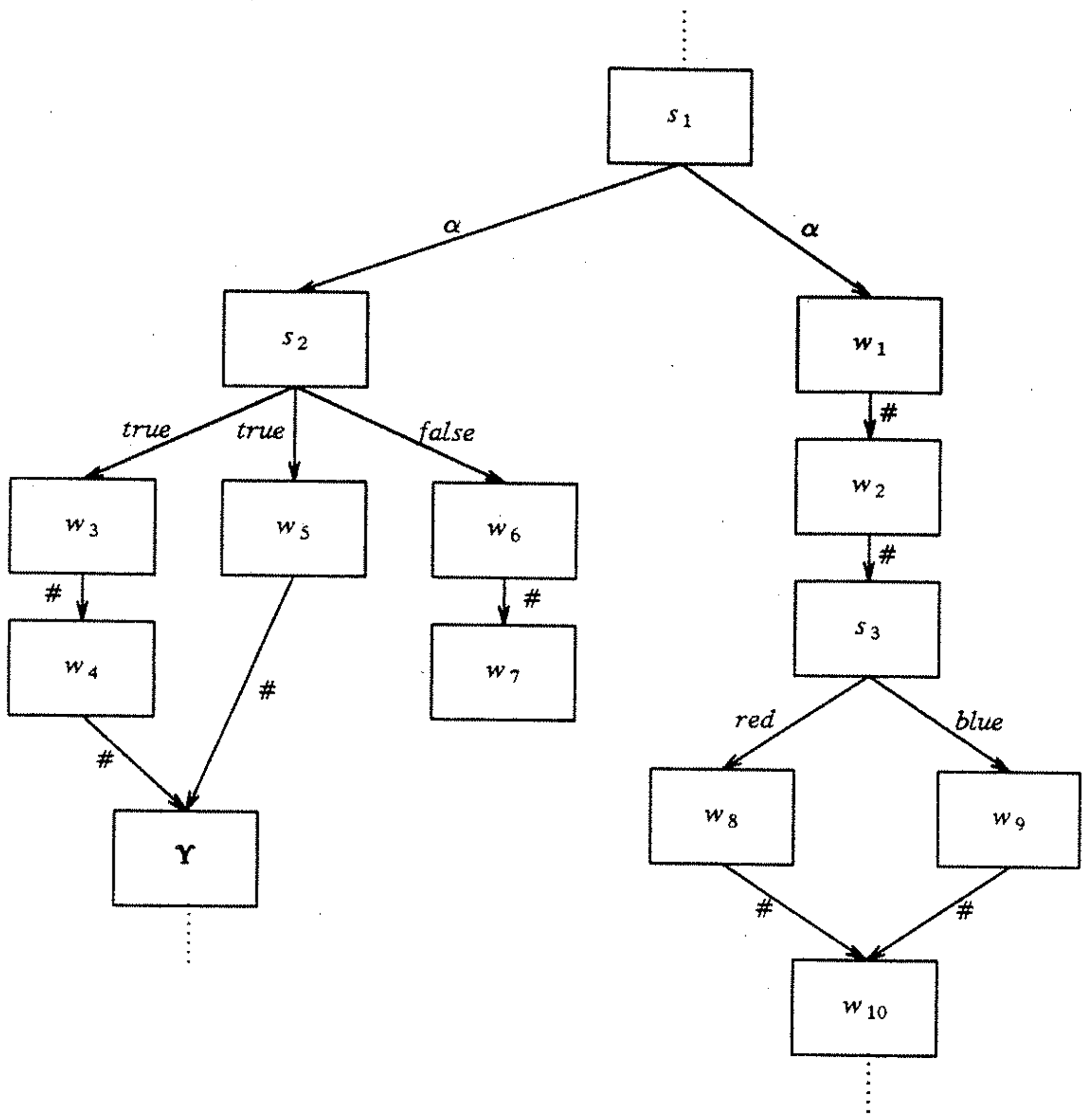


Figure 4.1 Parallel flow graph

sequence as a single node. The label on such a node is b_i to indicate a block of procedure calls rather than a single procedure call. This notation is illustrated in Figure 4.2.

Execution of the flow graph proceeds from the initial node. An initial data state, represented by an h-graph, is assumed. If a node contains a procedure call, then the data state is altered as specified by the definition of the called procedure. If a node contains a selector, then the data state is consulted at the node selected and a choice of next node (or nodes) is made based on the value found there. Since separate arcs leaving a branch node need not have unique labels, several control paths may be concurrently started from such a node. Two or more parallel control paths can be created in this way. If a node contains a join **Υ**, then synchronization of the potentially many incoming concurrent control paths is performed, and a single control path continues from the node.

An elaboration on arc following is in order. On completion of the action at any PFG node, the next nodes in the execution sequence are determined by following one or more

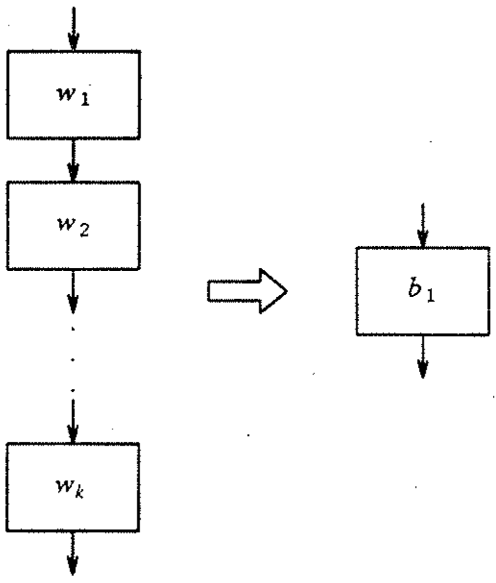


Figure 4.2 Sequence of procedure call nodes as a single node

of the arcs (if any) leaving the current node. The atom that identifies the arcs to follow is found in one of two ways, depending on whether the current node is a branch or not. If the current node contains a branch selector, then the appropriate atom is identified as previously stated: the value of the node chosen by the selector identifies the arc or arcs to follow. If the current node contains either a procedure call or a join, though, the desired atom is simply defined to be #. Recall that procedure call and join nodes are restricted to having no more than one out-arc each, which must be labeled with the *null* atom #. That single arc, if present, is then followed. Since branch nodes are the only nodes allowed to have multiple out-arcs, they are the only points in a PFG at which concurrent control paths can be created.

If no arc bearing the specified atom as label is found leaving a node, then the control path through that node expires: execution does not continue from the node. Execution of the entire PFG terminates when all individual control paths expire.

Since a PFG has a single initial node, the execution of a PFG always begins with one control path. When a branch node selector produces an atom that labels several out arcs, then concurrent control paths come into being. Subsequently, the progression of actions along each parallel control path is considered to be executing asynchronously and concurrently with the other parallel paths. Though the synchronization and merging of parallel paths is possible with **Y** nodes, it is not required. Two or more parallel paths may come together in a common segment of a PFG without being joined. Each path retains its separate identity and proceeds in turn to execute the PFG nodes in the common section. This feature, coupled with the fact that PFGs may be cyclic, allows a potentially unbounded number of parallel paths to be created in a computation. The number of such paths that can actually be executing at any time (as opposed to activated but waiting) is bounded, however, since the number of nodes in a PFG is finite.

Note that sequential computation is represented by a special form of the parallel flow graph, one in which labels on arcs leaving a branch node must be unique. Since no more than one arc may leave a procedure node, and since under this restriction no arc leaving a particular branch node may bear the same label as another arc leaving that node, at most one control path may proceed from each node in a sequential PFG. With only one initial node, no concurrent activity can then be generated. This simple and succinct restriction adds to the attractiveness of the theory as a unified computation model.

Though graphs are fairly well understood from flowcharting, we can go one step further in abstraction away from the model by indicating that a software designer need not even deal with graph structures in producing a program. As discussed in section 4.4 below, many standard programming language features are easily translated into parallel flow graphs (and hence to the component HG models). The transformation can occur either in intermediate steps, *i.e.*, text $\rightarrow$ PFG $\rightarrow$ HG models, or directly.

4.2. Extracting the model components from a PFG

Once a designer has produced a PFG which describes a desired computation, the component HG models may be derived automatically from the graph. All analysis is performed on the models, not on the parallel flow graph.

We first look at the constructions from a descriptive level and postpone a detailed presentation. To form the static program model, each basic block is produced from a node (or sequence of nodes) appearing at or between branch points and merge points in the PFG. For the control flow model, each node (or sequence of nodes) in the PFG produces a subnet component having a particular form, depending on the role that the PFG node performs. The Petri net components are then hooked together in the same manner as the PFG nodes.

Each basic block produced from a sequence of procedure call nodes in the PFG creates a place/transition pair (termed a *P/T component*) in the Petri net. The two are connected by a directed arc from the place to the transition. The P/T component is the basic net structure. If the place in such a component is empty, the arrival of a token represents the beginning of execution of an instance of the basic block associated with that place. The firing of the transition in the pair signals the end of that execution. A token arriving at an occupied place is treated as a request for an execution, which begins as soon as the transition in the P/T component fires to terminate the current instance.

Each branch node produces a *P/nT component*, composed of a place followed by n transitions where n is the number of distinct atoms appearing as labels on the arcs leaving the branch node in the PFG. Each arc connecting the place to a transition is associated (through the path selection function ρ) with a different one of these atoms. A transition has as many arcs leaving it as there are arcs bearing its atom as label leaving the branch node in the PFG.

Each PFG join node produces in the Petri net an *nP/T component*, composed of n places serving as input to a single transition. The number of arcs coming into the join node determines n; it represents the number of active parallel paths of control that are to be synchronized and merged. Figure 4.3 illustrates these constructions for a portion of a parallel flow graph. We now explain these procedures in more formal detail.

4.2.1. Building the static program model

The static program model, being a set of basic blocks, each of which is a sequence of procedure calls, is formed from the values of nodes in the PFG according to the arcs connecting them. Each branch node produces a basic block, representing the procedure calls required to evaluate the selector expression that produces the atom controlling branching. We let each join node create a basic block as well, to contain the procedure calls required to perform synchronization and control path deletion. The remaining basic blocks are formed by combining sequences of procedure call nodes. A new sequence starts with each procedure call node immediately following a branch node or a join node; a new sequence

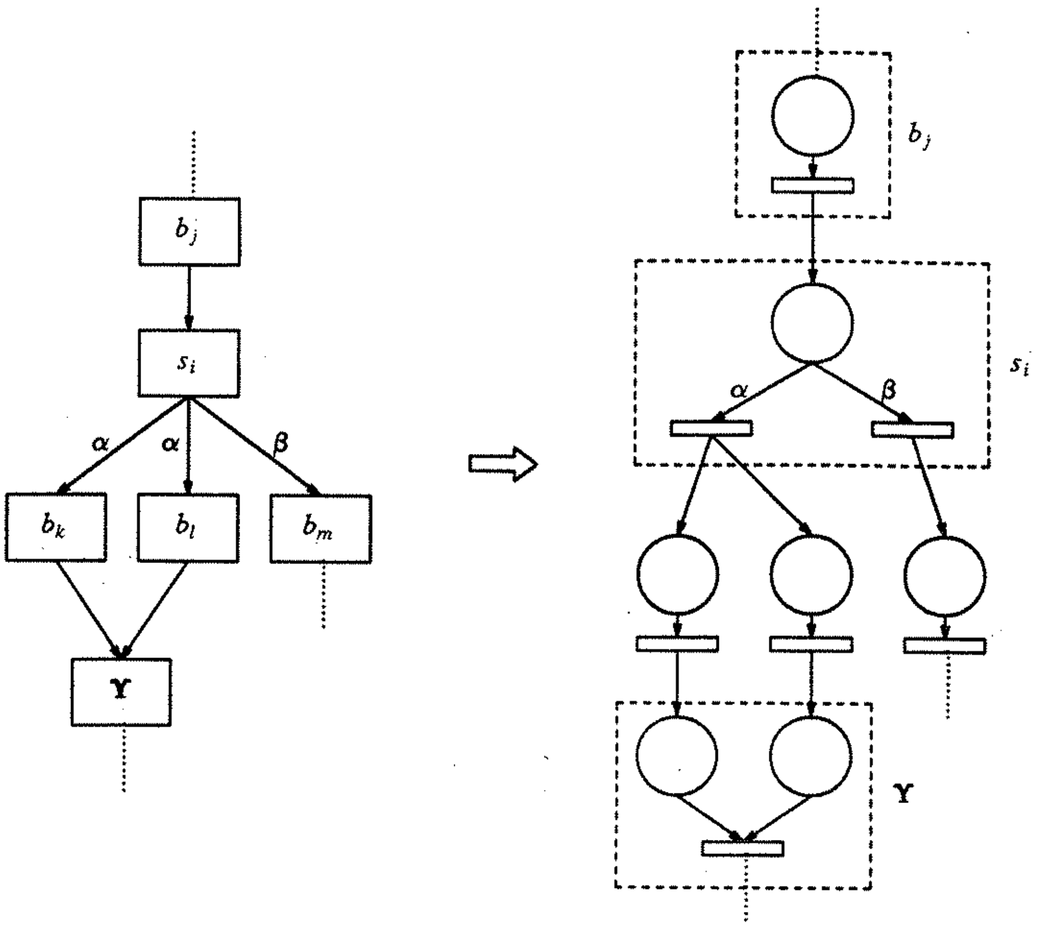


Figure 4.3 Creating P/T, P/nT, and nP/T components for PFG nodes

also must start with any procedure call node having more than one input arc, and with the initial node η'. A sequence then includes (in PFG order) all procedure call nodes on the path between its initiating node and a branch node, a join node, the initial node η', or a procedure call node with multiple input arcs. The basic block produced by a subsequence $\eta_1, \cdots, \eta_j$ of PFG nodes is $\langle w_1, \cdots, w_j \rangle = \langle V(\eta_1), \cdots, V(\eta_j) \rangle$. This basic block becomes one of the elements of the set B of basic blocks in the static program model. An exhaustive search of the PFG will produce all such basic blocks, constructing the set such that $B = \{b_1, b_2, \cdots, b_m\}$. Ordering of the basic blocks in the set is irrelevant to their respective locations in the PFG.

The set of named procedures NP in the static program model is constructed as simply the collection of all procedures named in the procedure calls in the PFG. Since only procedure calls, selectors, and **Υ** exist as values of PFG nodes, the set is easily isolated. The set S likewise is simply the collection of all selectors that appear in the PFG, and it is equally easy to construct. Algorithm 4.1 summarizes the procedure for constructing the static program model from a parallel flow graph.

The algorithm we present for constructing the static program model seeks to build the largest possible basic blocks, with the overall goal of producing from a procedure the smallest possible Petri net control flow model. The smaller the Petri net, the more tractable are the analysis algorithms, many of which require time and space exponential in the size of the net. Since the procedure calls between multi-arc nodes in the PFG are in a sequence, we produce a single block from them. However, if two blocks were produced from such a PFG structure the execution order of the component procedure calls would be the same. More generally, the individual basic blocks may be as small as that produced by a single procedure call node or as large as that produced by the unsubdivided sequence of PFG nodes as just described. This decision is a design choice and depends on the analysis required of the system. Our algorithm makes the choice not to subdivide, but there are situations in which subdivision may be desirable. For example, in representing a pipelined computation the components of the pipe must be separate basic blocks in order to have several of them concurrently executable (see section 4.4.5). Such situations require a modified version of the construction algorithm for the static program model.

Note that the construction algorithm does not specify the contents of the basic blocks created from branch and join nodes. One block for each such node is included in the set B in the static program model. Any selectors and procedures referenced in them are included in S and NP, respectively. The basic block formed from a branch node is

```
algorithm build_static_program_model ( $\phi$ )
--
--
-- The input argument $\phi=\langle g, V, \hat{\tau}\rangle$ is a PFG
--
-- The model is built in three pieces:
--    F : set of procedure names
--    S : set of selectors
--    B : set of basic blocks
--
-- This algorithm creates a basic block from each branch node, each join
-- node, and each maximal sequence of procedure call nodes; the PFG initial
-- node is handled first to ensure that it begins one of the basic blocks.
--
--
$F, S \leftarrow \varnothing$
for all nodes $\eta_i$
    assert $\neg$ seen($\eta_i$)
    add selectors in $V(\eta_i)$ to $S$
    if $V(\eta_i) \in W$
        add procedure named by $V(\eta_i)$ to $F$
$B \leftarrow \varnothing,\ j \leftarrow 0$
for all nodes $\eta_i$ ( $\eta_1 \leftarrow \eta'$ ) such that $\neg$ seen($\eta_i$)
    if $V(\eta_i) \in S \cup \{\Upsilon\}$
        assert seen($\eta_i$)
        $j \leftarrow$ succ($j$), $b_j \leftarrow \langle \eta_i \rangle$
        add $b_j$ to $B$
        for all nodes $\eta_t$ such that $\langle \eta_i, \eta_t, a\rangle \in E_\phi,\ a \in \Delta$
            $j \leftarrow$ succ($j$), $b_j \leftarrow \langle\,\rangle$
            while $\exists \eta_t$ and $V(\eta_t) \in W$ and $\eta_t$ has one input arc
                assert seen($\eta_t$)
                add $\eta_t$ onto $b_j$
                $\eta_t \leftarrow \eta_{next}$ such that $\langle \eta_t, \eta_{next}, \#\rangle \in E_\phi$
            add $b_j$ to $B$
    else if $\eta_i$ has multiple input arcs -- know here that $V(\eta_i) \in W$
        $j \leftarrow$ succ($j$), $b_j \leftarrow \langle\,\rangle$
        $\eta_t \leftarrow \eta_i$
        while $\exists \eta_t$ and $V(\eta_t) \in W$ and $\eta_t$ has one input arc
            assert seen($\eta_t$)
            add $\eta_t$ onto $b_j$
            $\eta_t \leftarrow \eta_{next}$ such that $\langle \eta_t, \eta_{next}, \#\rangle \in E_\phi$
        add $b_j$ to $B$
end build_static_program_model
```

Algorithm 4.1 Constructing the static program model

meant to provide a means to model the overhead encountered in activating concurrent control paths on the host machine; it should contain procedure calls to accomplish this task. The block for a join node should likewise contain procedure calls modeling the overhead in eliminating concurrent paths. Since these basic blocks represent system-specific information, defined outside the context of the PFG, they are not considered in detail here.

4.2.2. Building the control flow model

The control flow model is more complicated to extract from the parallel flow graph. A translation is performed from the PFG, guided by the node types and their arc interconnections, into Petri sub-nets, termed *net components*. The components are connected together as dictated by the PFG arcs. We look first at the net component created for each type of PFG node, and then we describe the interconnections. As each component is developed, we describe its contribution to the functions τ, β, σ, ρ, and the initial marking μ_0 of the control flow model.

4.2.2.1. Coalescing sequences of procedure call nodes

Before generating any Petri net structures, the number of PFG entities to be handled is reduced. The sequences of individual procedure call nodes, each of which forms a basic block in the static program model, are coalesced into single *call sequence* nodes as illustrated in Figure 4.2. Since there is a one-to-one correspondence between these nodes and the sequences of calls which form basic blocks, the call sequence nodes can actually be created when the static program model is built. We shall treat these "group" nodes as notational conveniences, and assume that the individual procedure call nodes comprising a call sequence η are recorded and numbered as $\eta = \langle \eta^1, \cdots, \eta^k \rangle$, where there are k procedure call nodes represented in η. If a call sequence node contains the initial node of the original PFG, it is designated the initial node of this modified formulation of the PFG.

4.2.2.2. Petri net components from PFG nodes

After the call sequence nodes are created, the marked timed Petri net TP is constructed by creating a subnet component with a distinct form for each type of node encountered in the resulting PFG. The specific form of the component for each node η is dictated by the set of PFG arcs *leaving* η (and for join nodes only, the arcs *entering* η). In a second step, after all the PFG nodes and their out-arc sets have been considered, the Petri net components are interconnected according to the arcs *entering* their generating PFG nodes.

Call sequence nodes

For each call sequence node η in the PFG, create a transition t_η, a place p_η, and an arc e_η to connect the pair such that

$$I(t_\eta) = \{p_\eta\} \quad \text{and} \quad O(p_\eta) = \{t_\eta\}.$$

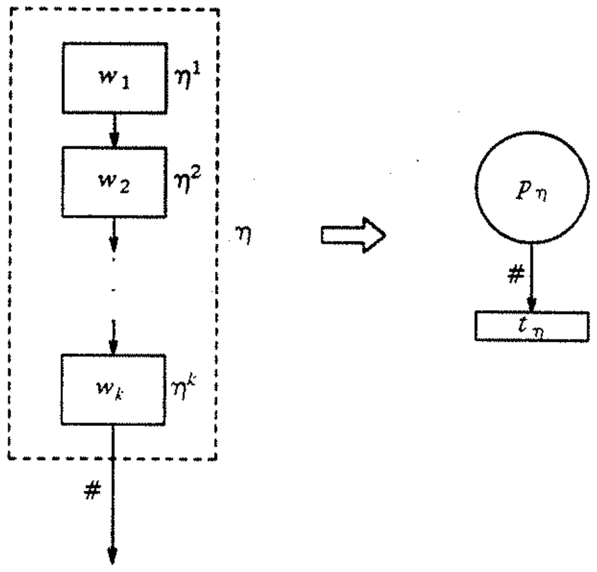


Figure 4.4 P/T Petri net component for a call sequence node

Note that the arc leaves p_η and enters t_η. Termed a *P/T component*, this Petri sub-net is illustrated by Figure 4.4. Let the label selector function σ associate with p_η a selector that designates a special node that always has the null atom # as its value. We assume here that the root node of the data state h-graph has this value, so use the selector "/". Let the path selection function ρ then designate t_η as the image of p_η and #. Let the basic block selection function β associate with p_η the basic block in the static program model which gave rise to the call sequence node η (call it b_η). Let the net timing τ associated with p_η be the sum of the PFG node timings of the procedure calls η^i comprising the sequence node η. Let the contribution of η to the initial marking μ_0 of the Petri net be that p_η is marked with a single token (and the token's age set to the place's duration) only if η is the initial node of the PFG. Formally, these constructions are expressed as follows:

- $\sigma(p_\eta) = /$
- $\rho(p_\eta, \#) = t_\eta$
- $\beta(p_\eta) = b_\eta$
- $\tau(p_\eta) = \sum_{i=1}^{k} \hat{\tau}(\eta^i)$, for k procedure calls in η
- $\mu_0(p_\eta) = \begin{cases} \langle 1, \tau(p_\eta) \rangle & \text{if } \eta \text{ is the initial node } \eta' \\ \langle 0, 0 \rangle & \text{otherwise} \end{cases}$

Branch (selector) nodes

For each branch node η in the PFG create a place p_η and n transitions t_η^i, $1 \leqslant i \leqslant n$ where n is the number of unique atom labels appearing on the arcs leaving η. Arcs e_η^i, $1 \leqslant i \leqslant n$, are created, one between p_η and each t_η^i, such that

$$I(t_\eta^i) = \{p_\eta\},\ 1 \leqslant i \leqslant n \quad \text{and} \quad O(p_\eta) = \{t_\eta^i \mid 1 \leqslant i \leqslant n\}.$$

Note that each arc is directed from the place to one of the transitions. Termed a *P/nT component*, this Petri sub-net is shown in Figure 4.5. Let the label selector function σ associate with p_η the selector which is the value of the PFG node η. Let the path

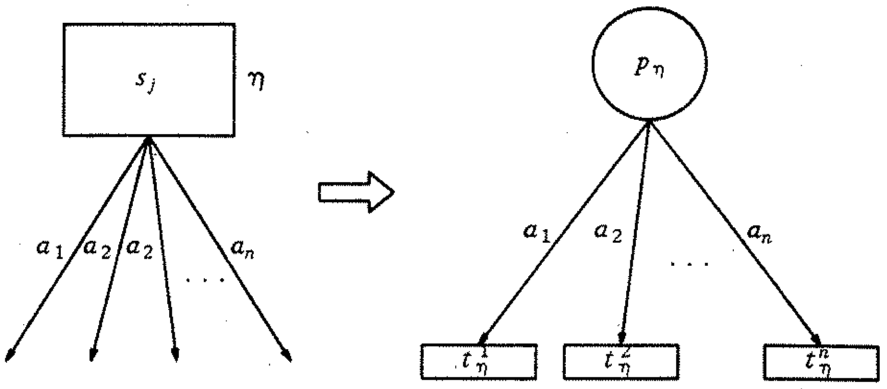


Figure 4.5 P/nT Petri net component for a branch node

selection function ρ designate each of the transitions t_η^i as the image of p_η and a different one of the n atom labels on the arcs leaving η. Let the basic block selection function β associate with p_η the basic block in the static program model which was formed from it (call it b_η). Let the net timing τ associated with p_η be simply the PFG timing of η. Let the contribution of η to the initial marking μ_0 of the Petri net be that p_η is marked with a single token (and the token's age set to the place's duration) only if η is the initial node of the PFG. Formally, these constructions are expressed as follows:

- $\sigma(p_\eta) = V(\eta)$
- $\rho(p_\eta, a_i) = t_\eta^i$, where $a_1, \cdots, a_n$ are the n unique atom labels on arcs leaving η
- $\beta(p_\eta) = b_\eta$
- $\tau(p_\eta) = \hat{\tau}(\eta)$
- $\mu_0(p_\eta) = \begin{cases} \langle 1, \tau(p_\eta)\rangle & \text{if } \eta \text{ is the initial node } \eta' \\ \langle 0, 0\rangle & \text{otherwise} \end{cases}$

Join nodes

For each join node η in the PFG create a transition t_η and n places p_η^i, $1 \leqslant i \leqslant n$ where n is the number of arcs entering η. Arcs e_η^i, $1 \leqslant i \leqslant n$, are created, one between each p_η^i and t_η, such that

$$I(t_\eta) = \{p_\eta^i \mid 1 \leqslant i \leqslant n\} \quad \text{and} \quad O(p_\eta^i) = \{t_\eta\},\ 1 \leqslant k \leqslant n.$$

Note that each arc is directed from one of the places to the transition. Termed an *nP/T component*, this Petri sub-net is shown in Figure 4.6. As with the call sequence nodes, let the label selector function σ associate with each p_η^i the selector "/" designating the root node of the data state h-graph, which is assumed to always have the null atom # as its value. Let the path selection function ρ then designate t_η as the image of each p_η^i and #. Let the basic block selection function β associate with each p_η^i a copy of the basic block in the static program model which was formed from η (call it b_η^i). Let the net timing τ associated with each p_η^i be simply the PFG timing of η. Since a join node may not be the initial node of a PFG, the contribution of η to the initial marking μ_0 of the Petri net is simply that each p_η^i is unmarked. Formally, these constructions are expressed as follows:

For $1 \leqslant i \leqslant n$, where η has n entering PFG arcs,

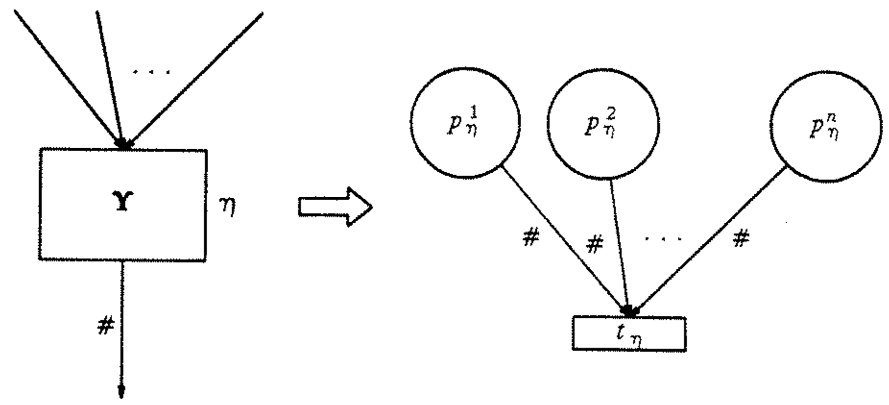


Figure 4.6 nP/T Petri net component for a join node

- $\sigma(p^i_\eta) = /$
- $\rho(p^i_\eta, \#) = t_\eta$
- $\beta(p^i_\eta) = b^i_\eta$
- $\tau(p^i_\eta) = \hat{\tau}(\eta)$
- $\mu_0(p^i_\eta) = \langle 0, 0 \rangle$

4.2.2.3. Arcs interconnecting Petri net components

Each arc created in the Petri net up to this point has been part of a net component and has been directed from a place to a transition. Next, Petri net arcs are created that correspond one-to-one with the arcs that enter and leave the nodes in the PFG, connecting the branch nodes, join nodes, and call sequence nodes. These net arcs serve to interconnect the net components and are directed from transitions to places. We look at the arcs generated by each PFG node η according to its type and the set of arcs that *enter* η. For a particular node η, this in-arc set is designated $E_\phi(\eta)$. Each arc $e_\phi \in E_\phi(\eta)$ is of the form $e_\phi = \langle \eta_i, \eta, a_i \rangle$, where $1 \leq i \leq |E_\phi(\eta)|$, indicating an arc from η_i to η, labeled a_i.

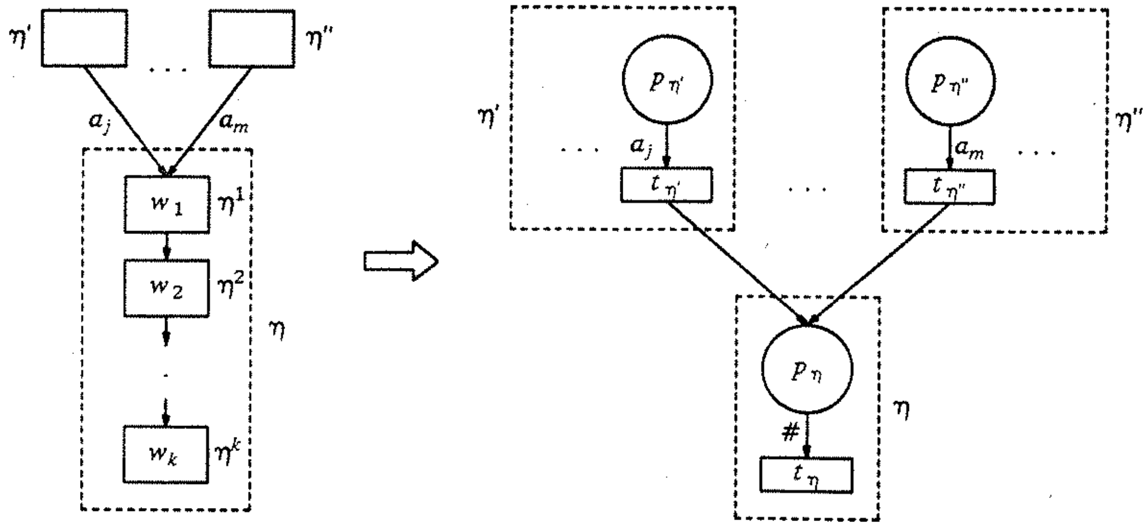


Figure 4.7 Arcs connecting components to a call sequence component

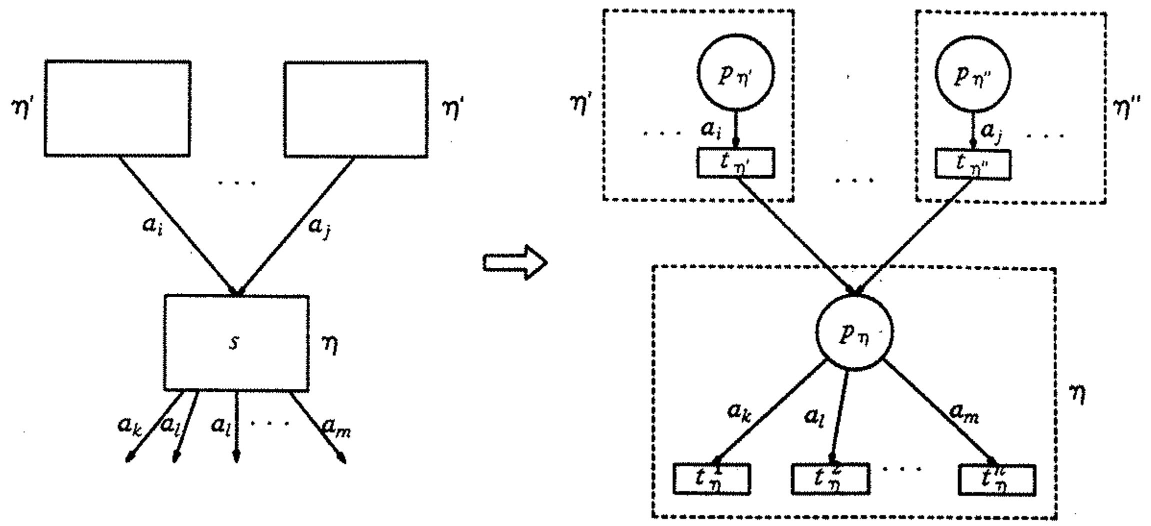


Figure 4.8 Arcs connecting components to a branch component

Call sequence nodes

Given a call sequence node η, for each $e_\phi \in E_\phi(\eta)$ construct arcs e_i, $1 \leqslant i \leqslant |E_\phi(\eta)|$, in the Petri net, each directed from the appropriately labeled transition in the component for η_i to the place p_η in the component for η. Formally, these arcs e_i are defined by:

Given that $t = \rho(p_{\eta_i}, a_i)$ for some place p_{η_i} in the Petri net component for η_i, let

- $t \in I(p_\eta)$
- $O(t) = \{p_\eta\}$

This construction is shown in Figure 4.7.

Branch (selector) nodes

Creation of arcs to connect arbitrary components to branch node components proceeds as for call sequence components. Both forms of component contain only one place, so all arcs coming in must connect to it. The formal expression of the construction is the same as in the previous section. Figure 4.8 illustrates the arcs created in this way.

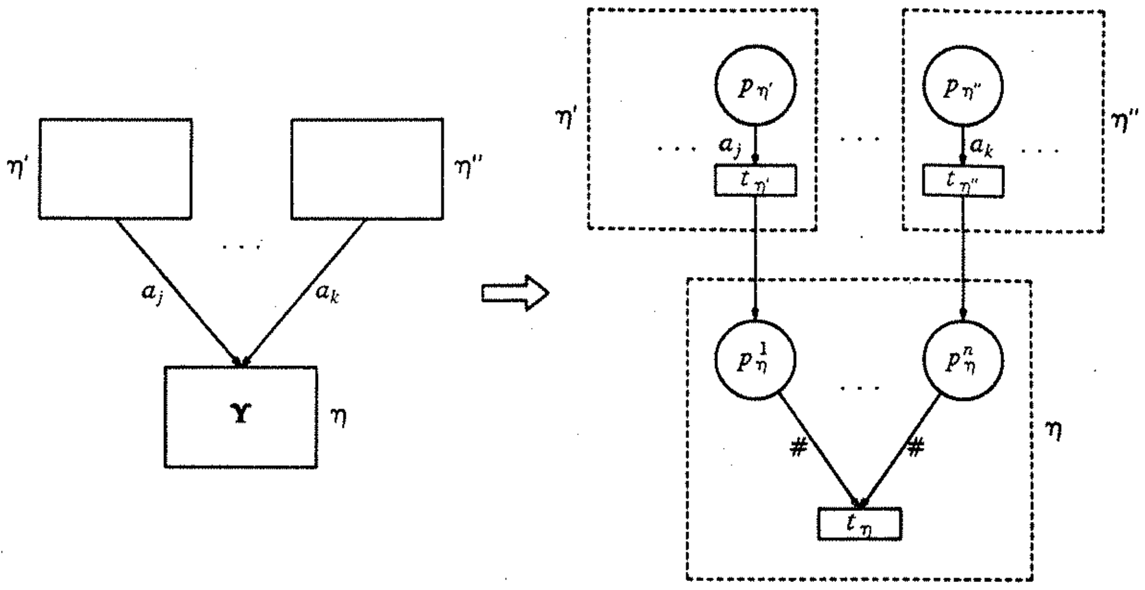


Figure 4.9 Arcs connecting components to a join component

Join nodes

The component for a join node η contains one place p_η^i for each of the $|E_\phi(\eta)|$ incoming arcs. For each $e_\phi \in E_\phi(\eta)$ construct arcs e_i, $1 \leqslant i \leqslant |E_\phi(\eta)|$, in the Petri net, each directed from the appropriately labeled transition in the component for η_i to a different one of the places in the component for η. Formally, these arcs e_i are defined by:

Given that $t = \rho(p_{\eta_i}, a_i)$ for some place p_{η_i} in the Petri net component for η_i, let

- $t = I(p_\eta^i)$
- $O(t) = \{p_\eta^i\}$

An example of this construction is shown in Figure 4.9.

The control flow model is thus created by a three-step process: coalesce procedure call nodes into call sequence nodes; create a Petri net component from each PFG node and its out-arc set; connect the net components together according to the in-arc set for each PFG node. The control functions β, σ, ρ, the initial marking μ_0, and the timing τ are all

```
algorithm build_control_flow_model ( $\phi$ )
--
-- The input argument $\phi = \langle g, V, \hat{\tau} \rangle$ is a PFG
-- The model consists of four pieces:
--    $TP = (P, T, I, O, \mu_0, \tau)$: the marked timed Petri net
--    $\beta$: the basic block selection function
--    $\sigma$: the label selector function
--    $\rho$: the path selection function
--
-- It is constructed in three steps:
--    (1) coalesce procedure call sequences into single PFG nodes
--    (2) construct sub-net components for each PFG node according to its out-arcs
--    (3) connect components according to node in-arcs
--
-- Assume that step (1) is done when the static program model is built, so that
-- the PFG has call sequence nodes instead of procedure call nodes
--
$P, T \leftarrow \varnothing$;   $\beta, \sigma, \rho, \tau \leftarrow$ undefined;   $\mu_0 \leftarrow \overline{\langle 0, 0 \rangle}$
--
-- construct the net components from the PFG nodes and out-arcs
--
for all PFG nodes $\eta$
    if $V(\eta) \in S$
        $P \leftarrow P \cup \{p\}$
        $\beta(p) \leftarrow b_\eta$
        $\sigma(p) \leftarrow V(\eta)$
        $\tau(p) \leftarrow \hat{\tau}(\eta)$
        if $\eta = \eta'$
            $\mu_0(p) \leftarrow \langle 1, \tau(p) \rangle$
        $O(p) \leftarrow \varnothing$
        for all atoms $a$ such that $\langle \eta, \eta', a \rangle \in E_\phi$
            $T \leftarrow T \cup \{t\}$
            $I(t) \leftarrow \{p\}$
            $O(p) \leftarrow O(p) \cup \{t\}$
            $\rho(p, a) \leftarrow t$
    else if $V(\eta) = \Upsilon$
        $T \leftarrow T \cup \{t\}$
        $I \leftarrow \varnothing$
        for all PFG arcs $e_\phi = \langle \eta', \eta, a \rangle$
            $P \leftarrow P \cup \{p\}$
            $I(t) \leftarrow I(t) \cup \{p\}$
            $O(p) \leftarrow \{t\}$
            $\beta(p) \leftarrow$ copy ( $b_\eta$ )
            $\sigma(p) \leftarrow /$
            $\rho(p, \#) \leftarrow t$
            $\tau(p) \leftarrow \hat{\tau}(\eta)$
```

Algorithm 4.2: Constructing the control flow model

```
    else — $\eta$ is a call sequence node
        $P \leftarrow P \cup \{p\}$
        $T \leftarrow T \cup \{t\}$
        $\beta(p) \leftarrow b_\eta$
        $\sigma(p) \leftarrow /$
        $\rho(p, \#) \leftarrow t$
        $\tau(p) \leftarrow \hat{\tau}(\eta)$
        if $\eta = \eta'$
            $\mu_0(p) \leftarrow \langle 1, \tau(p) \rangle$
—
— interconnect the net components according to the in-arcs of each PFG node
—
for all PFG nodes $\eta$
    if $V(\eta) = \Upsilon$
        for all PFG arcs $e_\phi^i = \langle \eta', \eta, a_i \rangle$
            $t \leftarrow \rho(p_{\eta'}, a_i)$ for some $p_{\eta'}$ in the component for $\eta'$
            $I(p_\eta^i) \leftarrow I(p_\eta^i) \cup \{t\}$
    else — $\eta$ is a call sequence node or a branch node
        for all PFG arcs $e_\phi^i = \langle \eta', \eta, a_i \rangle$
            $t \leftarrow \rho(p_{\eta'}, a_i)$ for some $p_{\eta'}$ in the component for $\eta'$
            $I(p_\eta) \leftarrow I(p_\eta) \cup \{t\}$
end build_control_flow_model
```

Algorithm 4.2 Constructing the control flow model (continued)

created when the net components are created. Algorithm 4.2 summarizes the procedure for constructing the control flow model from a parallel flow graph.

4.3. Modeling language constructs with PFGs

As presented in the previous chapter, in definition 3-19, the control flow model employs a general Petri net structure with no restrictions on the types of place-transition interconnections that may appear. Petri nets can be constructed that have no immediate interpretation in terms of software processes executing for some duration. Using parallel flow graphs for model design ensures that only well structured Petri net models will be produced, ones that have clear interpretations in terms of software systems. We have associated with each software unit (a basic block) in a procedure a place p_i in the Petri net model, for which the timing value $\tau(p_i)$ represents the duration of the software's execution. A token arriving at such a place signals the start of execution of the associated basic block. The token must reside at the place for a time; when it is fully aged and any directly following transitions have fired, the block of procedure calls is viewed as having completed its execution.

One interesting note about token aging is that the association of blocks of procedure calls with net components allows a measurement of waiting times if execution should be delayed for synchronization or for some other reason. Aging of a token in the control flow model is accomplished by starting a token at a place p_i with an age equal to the time for the place, $\tau_i = \tau(p_i)$. The age is decremented at each state change until the age is one, at which point the token is allowed to enable any following transitions. A fully aged token in an input place may not, however, be sufficient to cause a transition to fire. Consider, for instance, the Petri net structure created by a join node: several places serve as inputs to a single transition, and each place must contain a fully aged token before that transition can fire. In such a case, the token continues to age past one, attaining a more negative age value with each subsequent state change in which the transition does not fire. Thus, a place with a negative age component in the net marking indicates that waiting is occurring, and the magnitude of the age indicates the number of time units that the associated basic block has been delayed.

In a PFG structure, initiation of concurrent activity is limited to branch points, and these have a particular Petri net representation. Synchronization of concurrent activity occurs at PFG join nodes and at procedure call nodes with multiple input arcs, each of which creates a distinct form of Petri net component. A discussion of mutual exclusion structures in these nets appears in the following chapter. Using a PFG, then, to form a Petri net creates a net structure that has easily identifiable features, each with a well-defined purpose in terms of the modeled software system. In particular, we know that in the marked, timed Petri nets produced from PFGs,

- a place with multiple inputs can occur in any part of the net, for it represents control passing to the start of a block of procedure calls;
- a place with multiple outputs can only occur at a branch point or at a place that controls mutual exclusion;
- a transition with multiple inputs can only occur at a join point or at the beginning of a mutually excluded section of the network;
- a transition with multiple outputs can only occur at a branch point at which concurrent activity is being initiated or at the end of a mutually excluded section.

The Petri net structures created from PFGs are of the class termed *free choice*, introduced by Hack [26]. Any transition t_i in a free choice Petri net that is in *conflict* with another transition t_j, that is, that shares one or more input places with t_j, has *only* a single place p in its input set $I(t)$. The choice of which transition to fire from a conflicting group is unconstrained by places other than p having to be marked. Hence it is a *free* choice. Hack provides necessary and sufficient conditions for a free choice Petri net to be *live* and *safe*. A net is *live* if no marking creates a deadlock, that is, if execution from *any* state will eventually enable every net transition. A net is *safe* if no place ever contains more than one token. The problem of determining if a net is live is not known to be decidable for general Petri nets. The safeness problem is decidable for general nets, using the reachability tree.

It is easily seen that PFGs produce free choice nets. The only time that arcs from places to transitions are created is when net components are made from PFG nodes. The only Petri net component with multiple transitions is that of a branch node, the P/nT component. A P/nT component has only one place in it, and all n of its transitions share that place as input. No arcs are created between components unless they are directed from a transition to a place. Hence, only free choice structures are created in the Petri net produced from a PFG.

When conflicts on shared data are detected, the alterations required to create mutual exclusion in the control flow model move the resulting net structures out of the free choice class into the class of general nets. These procedures are described in the following chapter. If no conflicts are discovered, then the Petri net is unaltered and remains free choice.

A Petri net produced by using a PFG has a distinct behavior in terms of concurrency. Paths of control in a computation are represented by tokens in the net. Since a PFG has but one initial node, execution always begins with a single control path. At a branch point a single path can be expanded into a fixed number of concurrently executing paths by the selection of a transition with multiple out-arcs. In a completely acyclic net, then, the total number of parallel control paths is bounded by the total number of tokens that can exist in the net at any point in execution. The addition of cycles in a net allows creation of a potentially unbounded number of paths, depending on the data values obtained during execution. Once a token is created, the path of control it represents remains in existence until one of two events takes place:

- it enables a transition with no out-arcs, and is absorbed when the transition fires;
- it arrives at a place in a join component, and is coalesced along with other paths into one path when the transition of the join component fires; the new path continues on from the join as a single token.

The existence of concurrent control paths creates concurrent activity in the data state of a procedure. Since basic block executions are associated with the aging of tokens in places in the Petri net, concurrently existing tokens (control paths) in separate places represent two or more basic blocks affecting the data state simultaneously. If their accesses overlap, the potential for destructive interference exists. Moreover, reading of locations in the data state occurs at branch points, when tokens arrive and the associated selectors are evaluated. This reading may overlap with other activity on the same locations. The remaining portions of this section deal only with the concurrent control aspects of PFG modeled computations. The issues of concurrent access to the data state are discussed in the final section of this chapter, and in the next chapter.

Given that well-structured Petri net models can be created automatically from PFG structures, we can model the concurrency constructs in programming languages with the unified PFG notation rather than forming the individual components of the software system model in an *ad hoc* fashion. A software tool can then perform the translation from the PFG to the HG software system model prior to analysis. To illustrate the flexibility of the PFG, we describe in the following sections several well known control constructs from existing programming languages, as well as some that are not available in languages. Included are the sequential selection, the process spawn, the fork-and-join, the multi-way cobegin, the pipeline, and barrier synchronization.

4.3.1. Sequential selection

As a reference point we include in this discussion the standard selection structure, as might be represented by a two-way conditional statement (*if-then-else*) or a multi-way branch (*case*) in an imperative programming language. Figure 4.10 illustrates the PFG

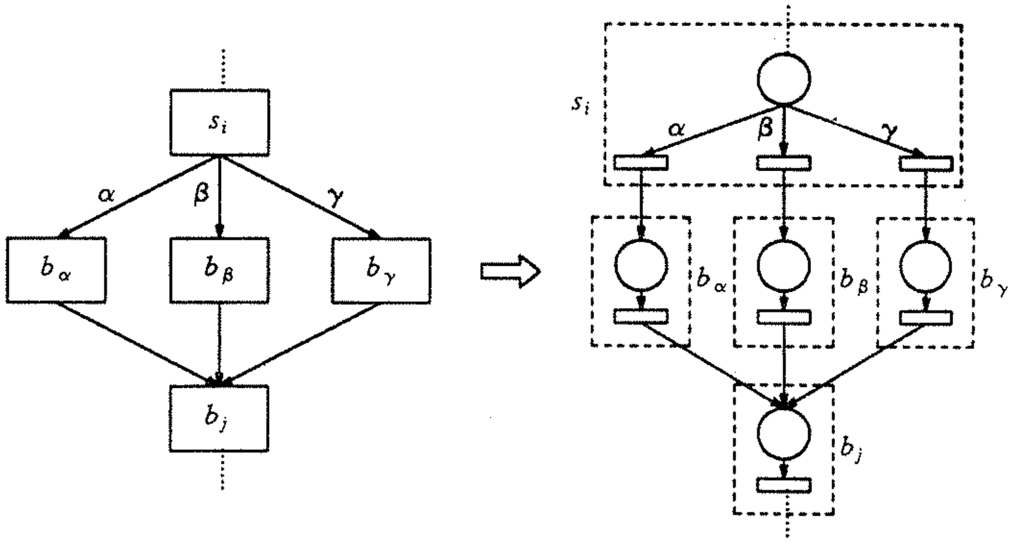


Figure 4.10 Sequential selection, multiple choices

representation and the Petri net structure formed from the PFG. The branch node creates in the Petri net a place with multiple output transitions, but the unique arc labeling indicates that only one of the transitions will ever fire at a time, so no concurrency is created. Hence no joining is required at the block that closes the selection structure.

4.3.2. Spawn processes

A basic construct for creating concurrent control paths in a computation is the process *spawn*, a facility in which several concurrent processes are started and the spawning agent continues its execution without waiting for them to complete. The UNIX operating system [59] offers the process spawn as its major means of creating concurrency. The PFG representation of such a structure is pictured in Figure 4.11 along with the Petri net structure that is produced from the PFG. In a sense, this structure is similar to the selection structure, but there is only one alternative to choose. That alternative then creates multiple control paths. Notice that the P/T pairs for the spawned processes have no arcs leaving their transitions. After the tokens age fully and the transitions fire, the indivi-

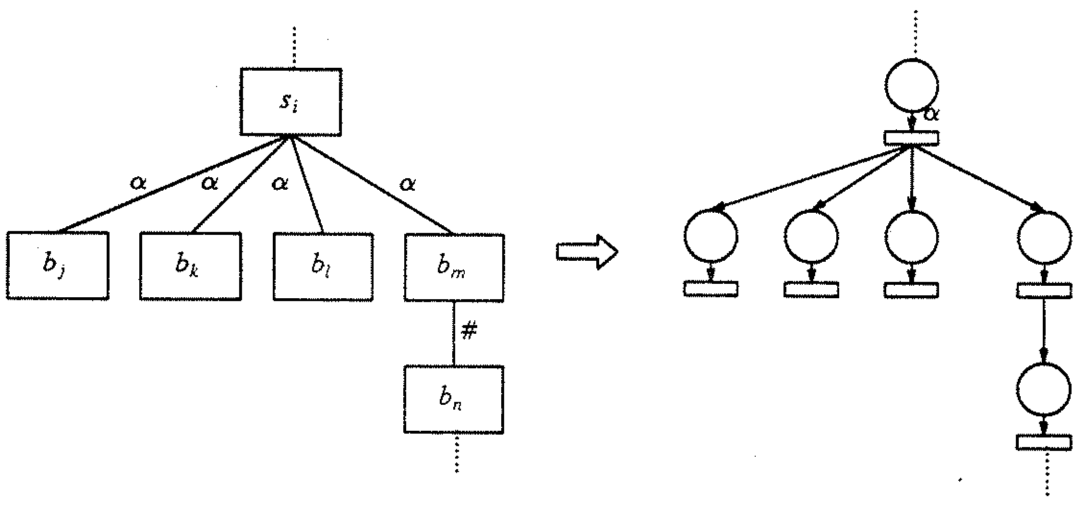


Figure 4.11 Spawn Processes

dual control paths expire, indicating that the termination of a spawned process does not trigger any further activity. There is no synchronization of any of the actions in this structure.

4.3.3. Fork and join

The *fork-and-join* structure is present in several languages. Gypsy [24] has such a feature, where it is termed a *cobegin*. This construct is similar to the process spawn structure, with the addition of synchronization. Several concurrent process executions are created, but eventually the individual control paths must merge back to a single path. Figure 4.12 illustrates this construct. Notice the inclusion of the join node to force syn-

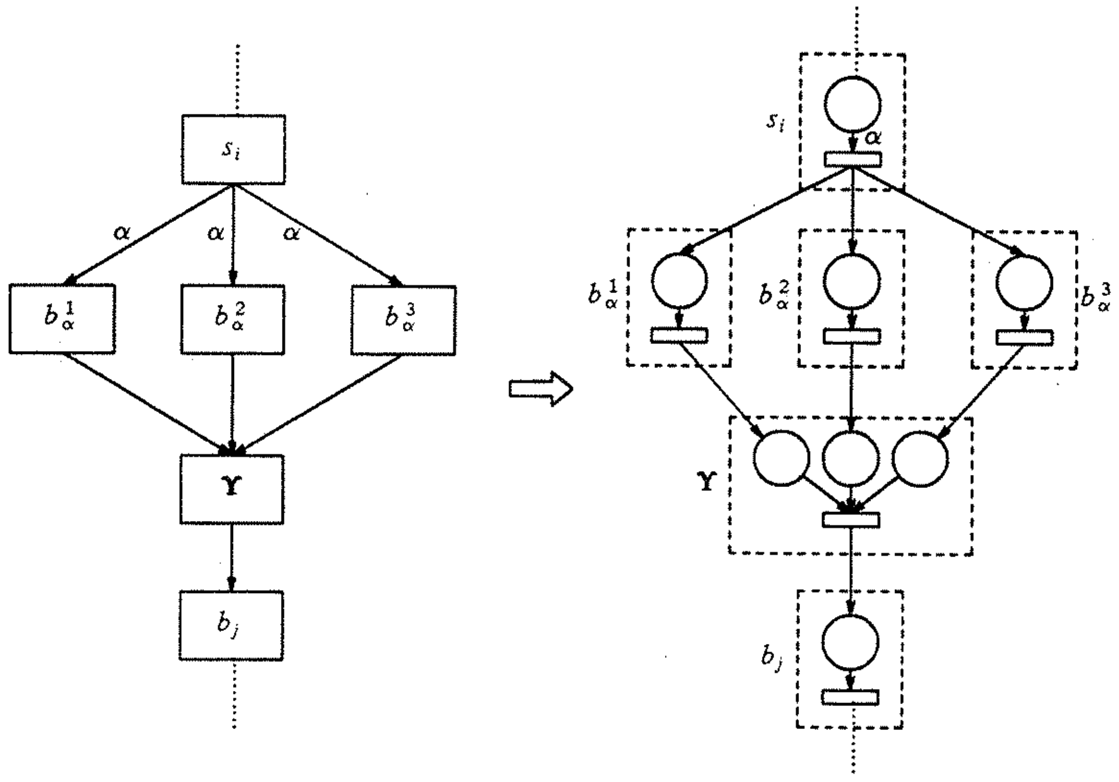


Figure 4.12 Fork and join (cobegin)

chronization of the separate process executions and to merge the multiple control paths back to the single path of the spawning agent.

4.3.4. Multi-way cobegin

Figure 4.13 shows a generalization of the fork-and-join that is easily expressed in the PFG notation. Different groups of concurrent processes may be initiated depending on the atom selected by the expression controlling the branching. In a sense this is also a generalization of the selection structure, in which one of several alternatives is chosen, but each alternative here may involve creation of concurrent control paths. Each alternative presents a complete cobegin structure in that the parallel paths are synchronized and

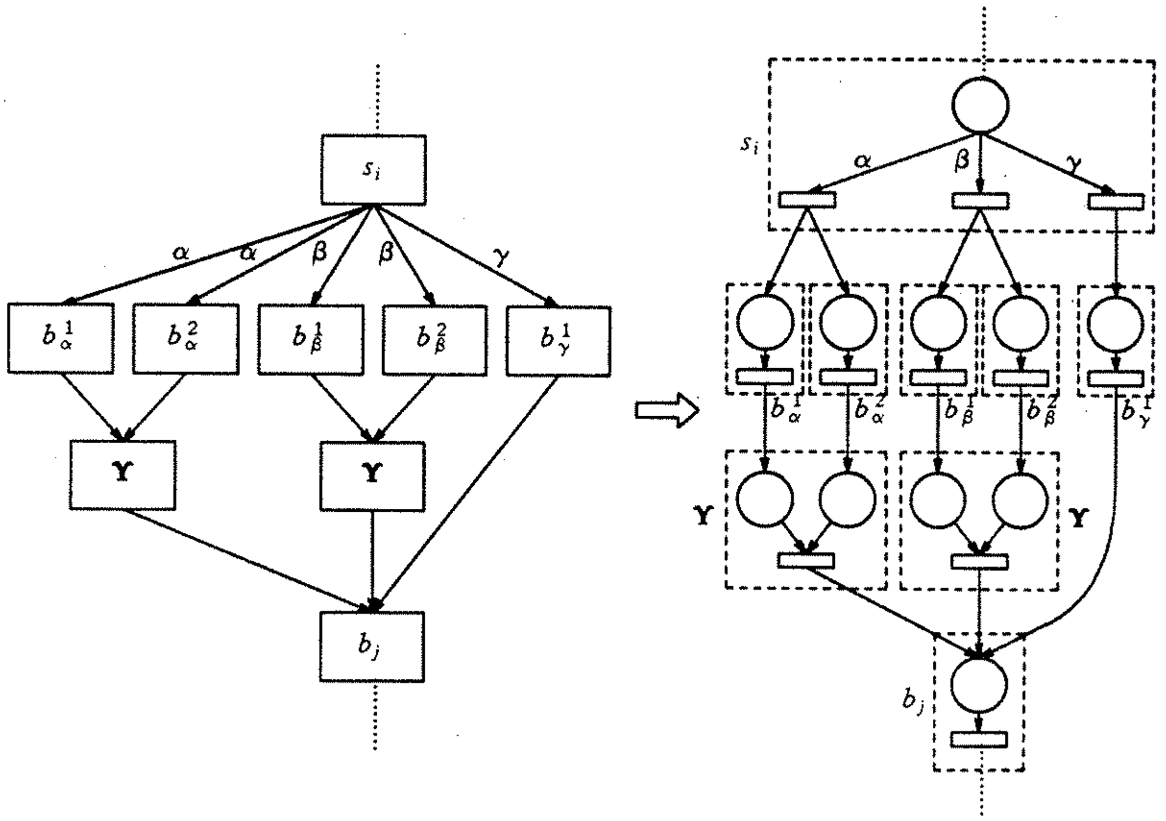


Figure 4.13 Multi-way cobegin

merged before continuing on to the block following the structure.

4.3.5. Process pipeline

Another common concurrency construct is the pipelined computation, shown in Figure 4.14. A branch node creates several concurrent control paths. Rather than run them to a join node (as in the cobegin), which coalesces into a single path, the concurrent executions are fed into a sequence of basic blocks. All paths continue to exist separately, but they now include some common blocks of procedure calls. Moreover, the semantics of

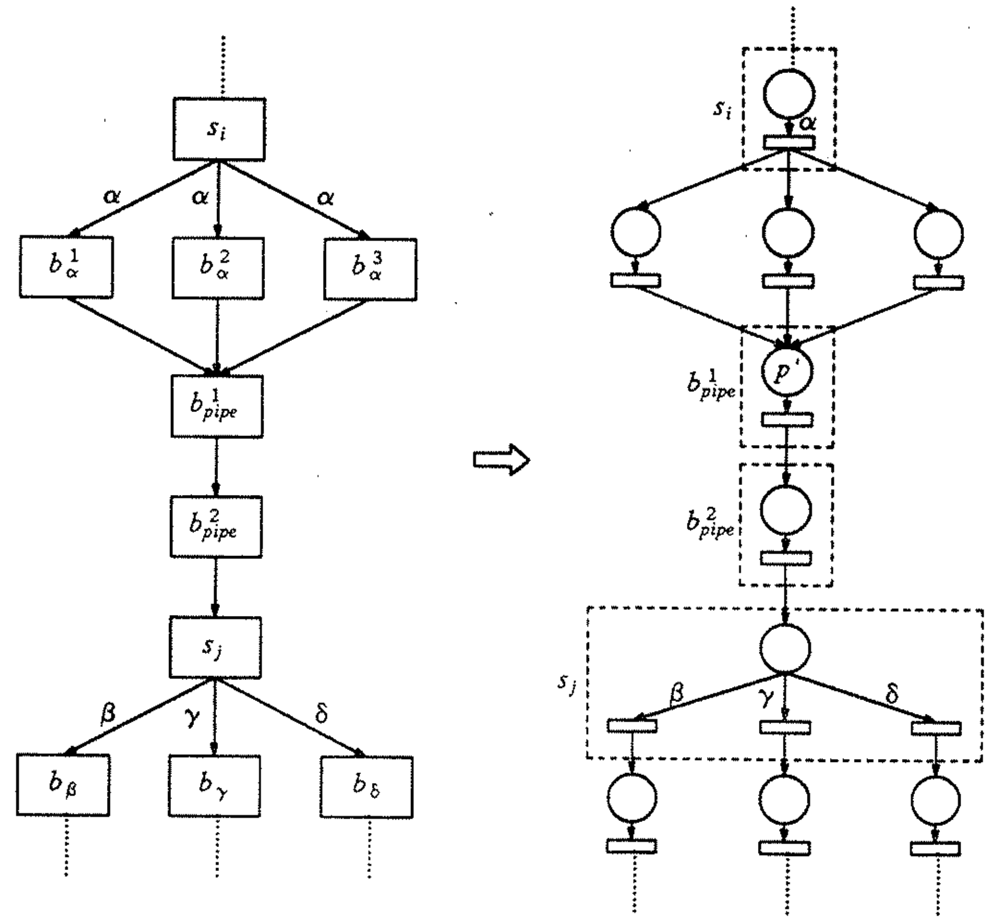


Figure 4.14 Pipelined computation

the Petri net execution rule force the multiple instances of these common blocks to be executed one-at-a-time rather than simultaneously. As execution on one of the control paths finishes the first pipeline block and moves on to the second, another execution of the first block will begin on one of the remaining control paths waiting at the head of the pipe. In the Petri net each control path is represented by a token, which must wait at the place representing the first basic block in the pipe until earlier tokens have moved on. Eventually each concurrent path will execute all blocks in the pipe with the indicated sequencing. At the end of the pipeline shown is a branch node to separate the control paths leaving the common blocks back into a fully concurrent structure.

4.3.6. Arbitrary structures

Figure 4.15 shows an example of the arbitrary control structures that can be constructed and studied using the PFG notation, the goal being to identify novel and useful control structures that can then be given a syntax and incorporated into a language design. The one pictured incorporates elements of a multi-way process spawn and a pipe-

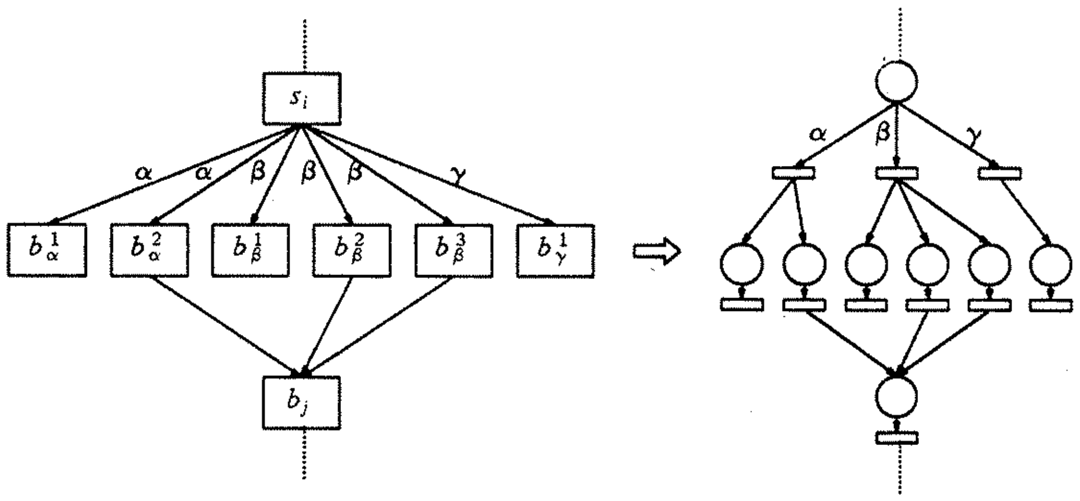


Figure 4.15 An arbitrary control structure

line. Some of the potential concurrent control paths expire; some pass on into the following block of procedure calls without being joined.

4.3.7. Barrier synchronization

Synchronization of concurrent tasks at a barrier is shown in Figure 4.16. The structure allows two independent tasks to synchronize their executions prior to a single

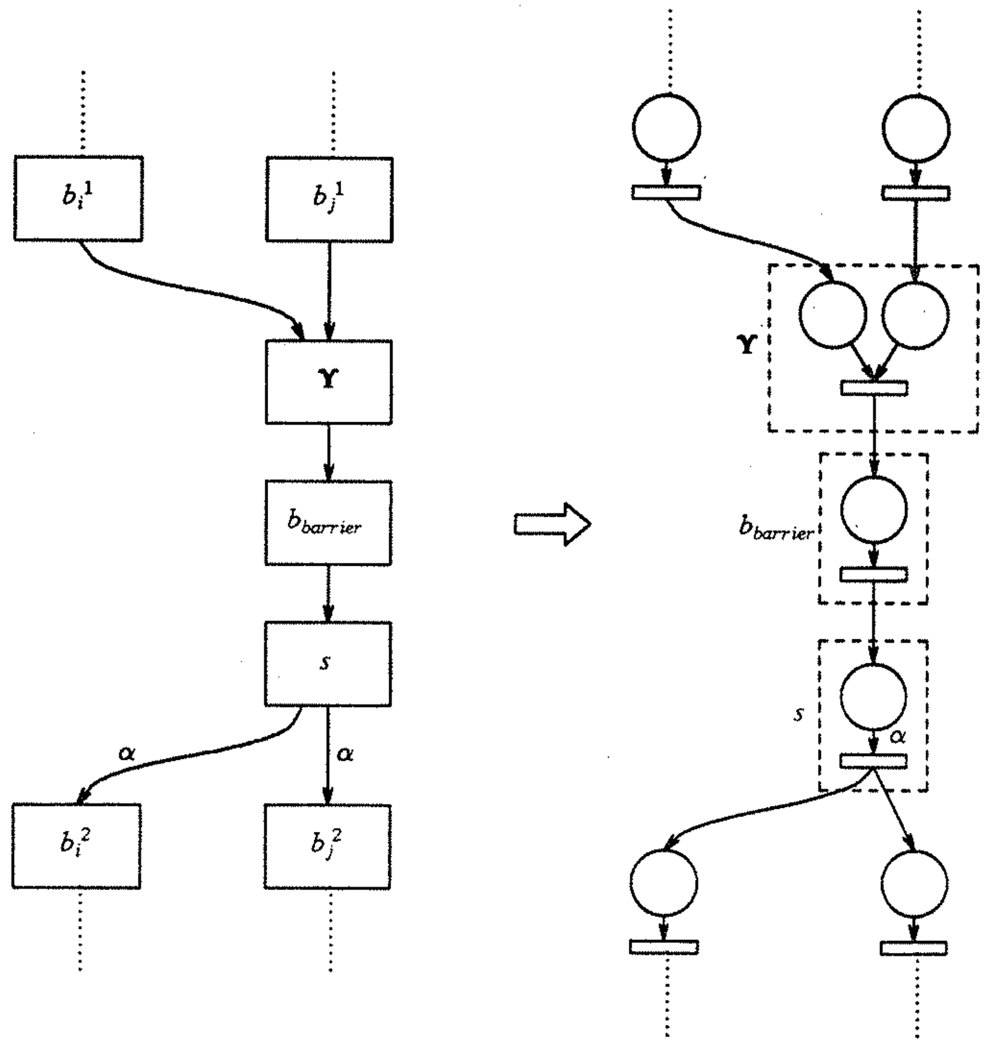


Figure 4.16 Barrier synchronization

instance of each block in the barrier being executed. After completion of the barrier, the synchronized control paths resume their original independent and concurrent executions. Forms of this structure appear in various existing languages, *e.g.*, Jordan's macro extensions to HEP Fortran [33] and the version of Pascal used on the Finite Element Machine [32, 16]. Though the *rendezvous* in Ada [2] is a barrier synchronization in terms of its control flow, the HG system model is insufficient to adequately model the full semantics of this form of task interaction. A barrier synchronization can only occur completely within a single PFG. This represents a single procedure, with a local data state shared by the concurrent blocks in the procedure. The rendezvous, however, allows no shared data other than the arguments passed during the interaction.

4.4. Modeling concurrent access to data in a PFG

The HG software system model offers two views of data: the distributed, global view of the system as a whole, and the shared, local view of each procedure. A system is a hierarchy of procedure calls, starting with the main procedure. There is a single data state for the main procedure, which is in a sense the global state for the system. Each procedure call causes the creation of a separate, local data area for execution of the called procedure. Argument values from the calling state are copied into the called local state on invocation, and values from the local state are copied back to the higher level state on termination of the call. In this sense, every procedure call is independent of all others. From a global perspective, the system is a completely distributed network of local data areas, each suitable for execution on a separate processing element with communication via message passing (of arguments and results).

From the point of view of an individual procedure, the local data area is being acted upon by potentially many concurrent actors. At each state change in a computation, several control paths may be active. Basic blocks execute concurrently as indicated by concurrently existing tokens in the Petri net, and selectors are evaluated at branch points,

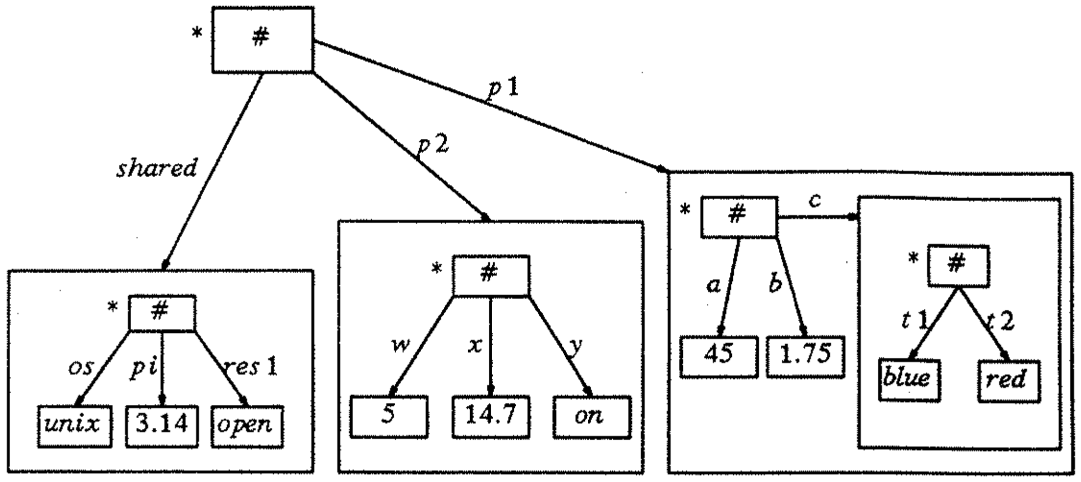


Figure 4.17 Partitioning of a procedure data state

possibly several at once. The procedure calls in a basic block are invoked in sequence, but an individual call in one block may be executing concurrently with a call in another block. Though individual procedure calls do not interfere with one another during their executions, when results are transmitted back to the calling environment they create potentially simultaneous reads and writes to locations in the local data state of the calling procedure. If two calls have any argument or result selectors in common, then they share local data. On this level, the HG model is a shared memory model.

A software modeler can exploit this duality when creating the local data area for a procedure. The selectors in a procedure model are either used completely within the blocks on one control path (task), or they are shared among two or more potentially concurrent tasks. The data state h-graph can be viewed as being partitioned into sub-h-graphs according to the usage of selectors by the concurrent tasks in a procedure, as shown in Figure 4.17. Each unshared partition is accessed by h-graph selectors having a common graph selector prefix. These selectors should appear only in one task of the procedure, meaning that they are not shared. For example, in the h-graph of Figure 4.17, selectors beginning with the graph selector "/p1" will only access data in this sub-h-graph:

```
[   [ # ]
       -a-> [ 45 ]
       -b-> [ 1.75 ]
       -c-> [ [ # ]
                -t1-> [ blue ]
                -t2-> [ red ]
            ]
]
```

The shared selectors also have a common prefix, since all shared locations lie in a single data partition. These selectors may appear scattered throughout many different concurrent tasks, indicating that they designate shared data. For example, again considering the h-graph in Figure 4.17, h-graph selectors having the graph selector "/shared" as prefix will select nodes in this sub-h-graph:

```
[ [ # ]
     -os-> [ unix ]
     -pi-> [ 3.14 ]
     -resl-> [ open ]
]
```

Such a data state organization lends itself to at least partial implementation of the model on a distributed, or other non-shared memory, architecture.

Note that these comments are organizational suggestions and not requirements of the modeling theory. A system designer is always free to treat the entire data state of a procedure as shared, or to employ an organization scheme that falls between the two extremes.

4.5. An extended example

As an illustration of the expressive capabilities of PFGs, an extended example is presented here and continued in subsequent chapters as analysis techniques are developed. Consider a software system to perform the three parallel tasks of continuously obtaining input data from an environment, solving a set of equations to produce some results, and transmitting the results back to the environment. An applicable scenario is obtaining coefficients for the quadratic formula from a producer, calculating the roots of the

```
[ # ]
  -const-> [ [ # ]
               -t-> [ true ]
               -f-> [ false ]
               -0-> [ 0.0 ]
               -2-> [ 2.0 ]
               -4-> [ 4.0 ]
           ]
  -shared-> [ [ # ]
                -cbuff-> [ [ # ]
                             -a-> [ 1.0 ]
                             -b-> [ 0.0 ]
                             -c-> [ 0.0 ]
                         ]
                -rbuff-> [ [ # ]
                             -r1-> [ 0.0 ]
                             -r2-> [ 0.0 ]
                         ]
                -quit-> [ false ]
            ]
  -task1-> [ [ # ]
               -ibuff-> [ [ # ]
                            -a-> [ 1.0 ]
                            -b-> [ 0.0 ]
                            -c-> [ 0.0 ]
                        ]
           ]
  -task2-> [ [ # ]
               -t1-> [ 0.0 ]
               -t2-> [ 0.0 ]
               -t3-> [ 0.0 ]
               -t4-> [ 0.0 ]
           ]
```

Figure 4.18 Initial data state h-graph for main procedure QUAD

quadratic formula using the *most recently* produced set of coefficients, and displaying the solution on a graphics device, again using only the most recently calculated data. A similar problem is presented by Bernstein [12] as being unsolvable in CSP without an extension to allow guarded output commands.

The software system model contains three procedure models: *quad* (the main procedure), *acquire*, and *display*. The procedures *acquire* and *display* represent the operation

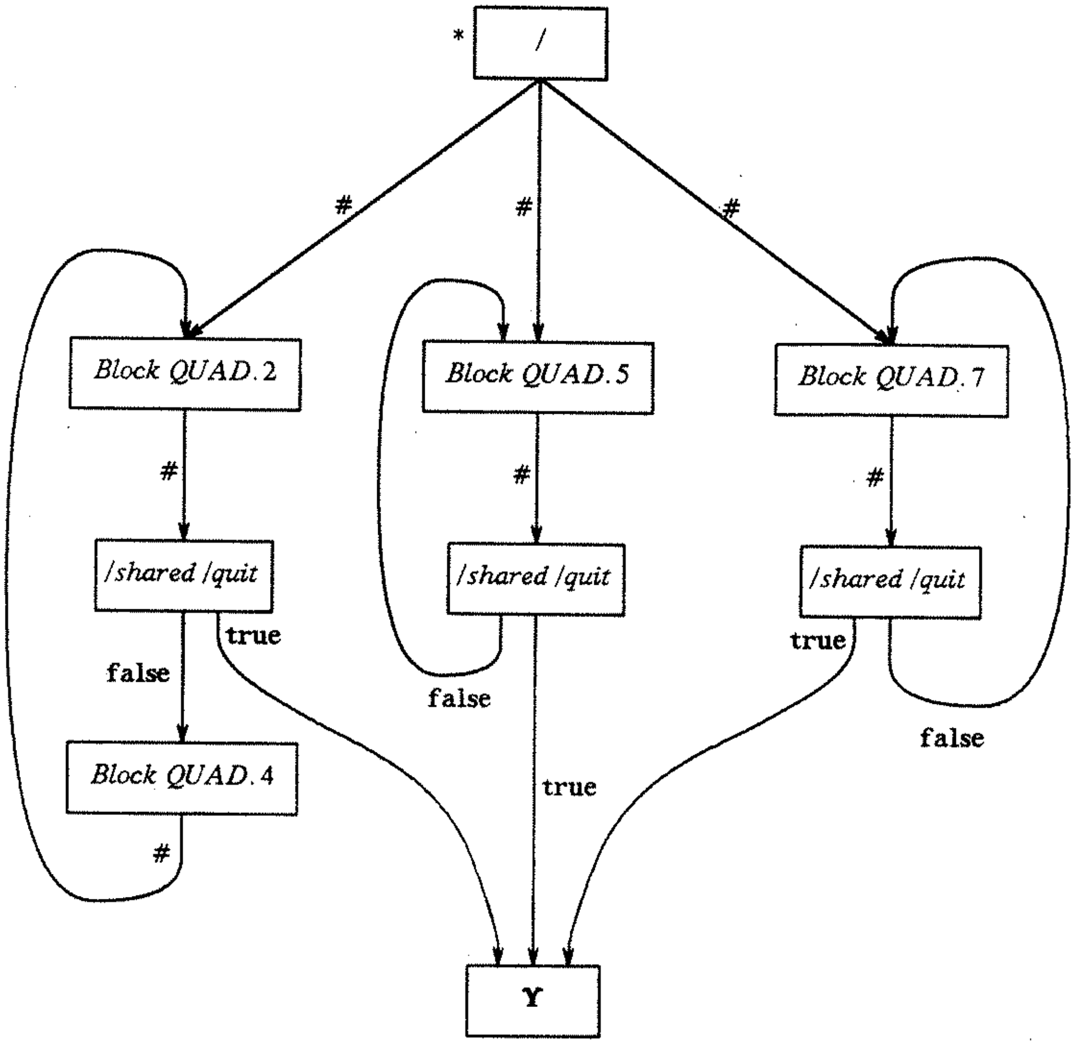


Figure 4.19 PFG structure for procedure QUAD

```
Block QUAD.2

/task1/ibuff/a, /task1/ibuff/b, /task1/ibuff/c := acquire()
/shared/quit := eq(/task1/ibuff/a, /const/0)
```

```
Block QUAD.4

/shared/cbuff/a, /shared/cbuff/b, /shared/cbuff/c
        := xfer(/task1/ibuff/a, /task1/ibuff/b, /task1/ibuff/c)
```

```
Block QUAD.5

/task2/t1 := mult(/const/4, /shared/cbuff/a)
/task2/t1 := mult(/task2/t1, /shared/cbuff/c)
/task2/t2 := sqr(/shared/cbuff/b)
/task2/t3 := neg(/shared/cbuff/b)
/task2/t4 := mult(/const/2, /shared/cbuff/a)
/task2/t2 := subt(/task2/t2, /task2/t1)
/task2/t2 := sqrt(/task2/t2)
/shared/rbuff/r1 := add(/task2/t3, /task2/t2)
/shared/rbuff/r2 := subt(/task2/t3, /task2/t2)
/shared/rbuff/r1 := div(/shared/rbuff/r1, /task2/t4)
/shared/rbuff/r2 := div(/shared/rbuff/r2, /task2/t4)
```

```
Block QUAD.7

/shared/rbuff/r1, /shared/rbuff/r2
        := display(/shared/rbuff/r1, /shared/rbuff/r2)
```

Figure 4.20 Basic blocks in PFG for QUAD

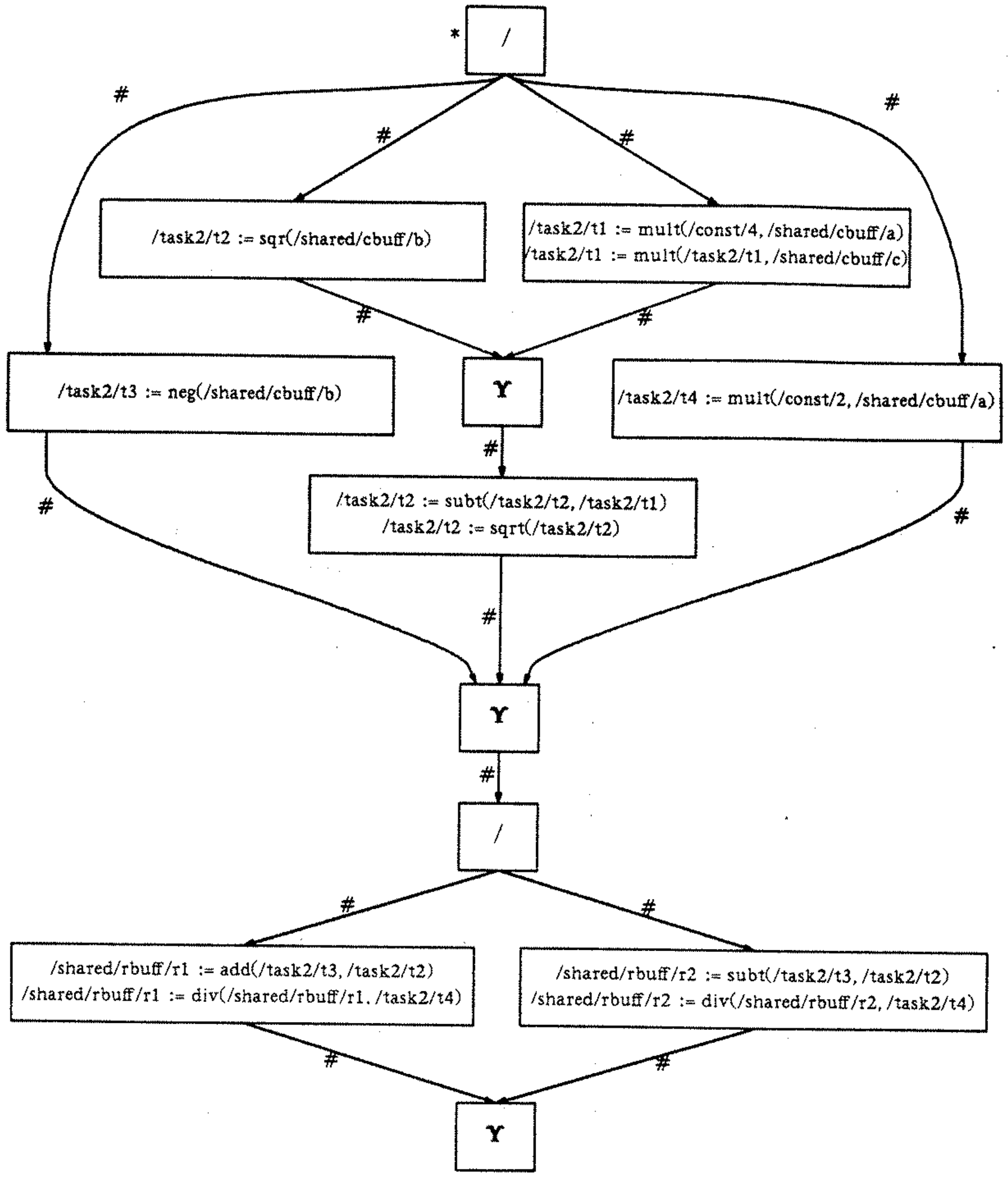

*
/
#
/task2/t2 := sqr(/shared/cbuff/b)
/task2/t1 := mult(/const/4, /shared/cbuff/a)
/task2/t1 := mult(/task2/t1, /shared/cbuff/c)
/task2/t3 := neg(/shared/cbuff/b)
Y
/task2/t4 := mult(/const/2, /shared/cbuff/a)
/task2/t2 := subt(/task2/t2, /task2/t1)
/task2/t2 := sqrt(/task2/t2)
/shared/rbuff/r1 := add(/task2/t3, /task2/t2)
/shared/rbuff/r1 := div(/shared/rbuff/r1, /task2/t4)
/shared/rbuff/r2 := subt(/task2/t3, /task2/t2)
/shared/rbuff/r2 := div(/shared/rbuff/r2, /task2/t4)

Figure 4.21 PFG structure for parallel form of block QUAD.5

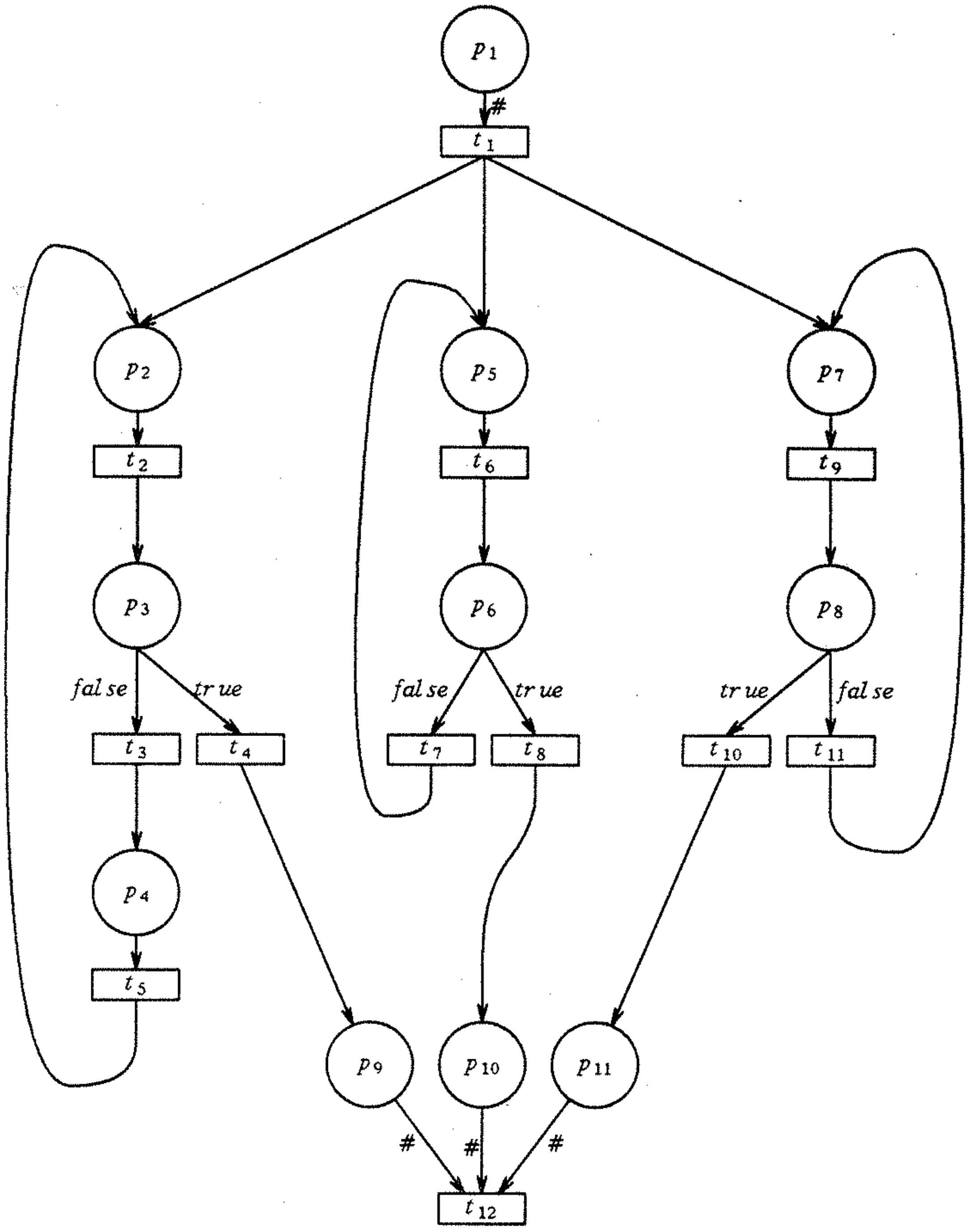


Figure 4.22 Timed Petri net for main procedure QUAD

place	β	σ	τ	μ_0
p_1	*Block NIL*	/	1	$\langle 1,1\rangle$
p_2	*Block QUAD.2*	/	1	$\langle 0,0\rangle$
p_3	*Block NIL*	*/shared /quit*	1	$\langle 0,0\rangle$
p_4	*Block QUAD.4*	/	1	$\langle 0,0\rangle$
p_5	*Block QUAD.5*	/	1	$\langle 0,0\rangle$
p_6	*Block NIL*	*/shared /quit*	1	$\langle 0,0\rangle$
p_7	*Block QUAD.7*	/	1	$\langle 0,0\rangle$
p_8	*Block NIL*	*/shared /quit*	1	$\langle 0,0\rangle$
p_9	*Block NIL*	/	1	$\langle 0,0\rangle$
p_{10}	*Block NIL*	/	1	$\langle 0,0\rangle$
p_{11}	*Block NIL*	/	1	$\langle 0,0\rangle$

place	atom	ρ
p_1	#	$\{t_1\}$
p_2	#	$\{t_2\}$
p_3	false	$\{t_3\}$
p_3	true	$\{t_4\}$
p_4	#	$\{t_5\}$
p_5	#	$\{t_6\}$
p_6	false	$\{t_7\}$
p_6	true	$\{t_8\}$
p_7	#	$\{t_9\}$
p_8	true	$\{t_{10}\}$
p_8	false	$\{t_{11}\}$
p_9	#	$\{t_{12}\}$
p_{10}	#	$\{t_{12}\}$
p_{11}	#	$\{t_{12}\}$

Figure 4.23 Functions in control flow model for QUAD

of a coefficient generator and an output device, respectively. Only the procedure *quad* is illustrated in detail here.

In addition to the procedure models, the system has several primitive procedures, some representing hardware instructions in an execution host: *ident*, *neg*, *sqr*, *sqrt*, *mult*, *subt*, *add*, and *div*. The procedure *ident* is a operation that simply returns its arguments as results. The others are standard arithmetic operations. Calls to these primitive procedures appear in the static program model for procedure *quad*.

The procedure *quad* is the main procedure for the system. The initial data state for *quad* is the h-graph shown in Figure 4.18. The structure of the PFG representing the computation of *quad* is shown in Figure 4.19. For simplicity of presentation, the individual procedure call nodes are not shown; the call sequence nodes resulting from the formation of basic blocks are pictured instead. Figure 4.20 shows the procedure calls comprising these call sequence nodes. The primitive procedures called in these blocks have no modeled structure in the system, but their effects on the data state are known. Note that in block QUAD.7, the assignment to nodes */shared/rbuff/r1* and */shared/rbuff/r2* is not necessary, but is included to make the conflict detection procedures in the next chapter more illustrative. The procedure *display* otherwise needs no result arguments.

Figure 4.21 presents an alternative structure for the procedure call sequence in block QUAD.5. The single block can be replaced by the portion of a PFG shown. In contrast with the macroscopic view of concurrency offered by the top level system PFG, this second formulation illustrates the fine-grained parallelism which is also achievable with the HG system model. In it, evaluation of the quadratic formula proceeds along several concurrent paths, one for each independent sub-expression. Paths are joined as dictated by the data dependencies of the formula, and the final two roots are computed concurrently. Since the numeric results are the same either way, for simplicity of presentation we develop the example using the single block form of QUAD.5 from Figure 4.20.

The control flow model for procedure *quad* consists of the timed Petri net shown in Figure 4.22 along with the functions shown in figure 4.23.

4.6. Summary

A graphical algorithm notation termed parallel flow graphs (PFGs) is introduced as a method of conveniently expressing a broad class of concurrent computations. A single PFG is meant to express the structure and potential behavior of a collection of concurrently executing blocks of procedure calls operating on a shared data space. A translation procedure is described from PFGs to the HG model formalism. HG models produced from PFGs offer a simpler structure for analysis than those produced in an *ad hoc* fashion, in that *free choice* Petri nets rather than general Petri nets are produced from PFGs.

Several well-known concurrent control structures from existing programming languages are expressed in the PFG notation to demonstrate the utility and generality of these modeling techniques. The structure of the data state for an HG system and an individual procedure model is discussed. An extended example is introduced that will carry over to subsequent chapters.

Chapter 5

CONFLICT RESOLUTION IN THE CONCURRENCY MODEL

We turn now to the problem of detecting and correcting conflicts among the basic blocks of procedure calls that may execute concurrently. The control flow model dictates the order and groupings in which basic blocks execute. If the control flow model calls for several blocks to be active simultaneously, we need to be sure that the concurrently executing instructions do not interfere with each other, for example by one block writing a variable between two successive reads of the variable in the second block. Should some of the concurrent blocks of a model be found to have conflicts, we then need a technique by which the blocks can be subdivided into smaller blocks such that the conflicts are eliminated and the computation may proceed in parallel with the correct mutual exclusion.

The parallel flow graph was introduced in the previous chapter as a unified representation of a static program model and a control flow model. In the ensuing discussion we will refer to a PFG ϕ and its representation as component HG models interchangeably, using the notation $\phi = \langle g, V, \hat{\tau} \rangle = \langle SP, CF \rangle$. In this chapter, then, we restrict the universe of static program models and control flow models under consideration to those that are produced from parallel flow graphs rather than formed in an *ad hoc* fashion. We further restrict the control flow models for which this conflict analysis is applicable to ones in which timing is not a consideration. For a control flow model $CF = \langle TP, \beta, \sigma, \rho \rangle$ in which $TP = \langle P, T, I, O, \mu_0, \tau \rangle$, this restriction is formally stated as

$$\forall p \in P [\tau(p) = 1]$$

and says that all places, and hence basic blocks, have an execution duration of unity.

This restriction is tantamount to ignoring the timing information in the model. The assumption is that at each state change we choose a set of basic blocks and concurrently execute each to completion before continuing with the next state change. In a sense, this treats a basic block as an atomic operation, which is not actually true to the system being modeled. The assumption is made to simplify the analysis. An extension to consideration of basic blocks overlapping in time is described in the following chapter.

5.1. Potential interference and concurrent conflict

The work of Bernstein [13] provides a starting point. His analysis enables one to discover when two blocks of code, one following the other in sequence, may be executed concurrently without risk of conflict. Our requirements are somewhat different. The two blocks of code in our analysis are originally to run in parallel, not in sequence. We need to discover the conflicts in data references, if any, that would prohibit this concurrency, and in the presence of such, to subdivide the blocks so that some portion of each may execute in lieu of the entire block.

The analysis is further complicated by the need to identify read/write conflicts not just on simple variables but on structured data as well. The discussion is motivated by the observation that if one procedure changes a field in, say, a record and another procedure changes the entire record as a unit, then the two are in conflict on shared data, even though textually the references to the data items are different.

In the HG model, the textual form of an h-graph selector gives an indication of the structure of the data involved with the function represented by the selector. If an h-graph selector s is composed of a single graph selector (*i.e.*, it has only a single, leading "/", as in "/a.b.c"), then it accesses one graph, and it designates one node within that graph by following various arcs (indicated by the "." notation in the graph selector). We consider this analogous to using a simple variable, treating the single involved graph as the collective local data state for a procedure model containing s. If an h-graph selector is composed of multiple graph selectors (*e.g.*, "/a.b/c"), then it involves more than one graph. This is analogous to accessing a structure within the local data state. The textual form of two selectors s_i and s_j can be used to detect any potential overlap of the nodes designated by $[[s_i]]$ and $[[s_j]]$. The concept of a selector prefix, as found in [64], provides the formal basis for textual reasoning about such conflicts.

Definition 5-1: Prefix of an h-graph selector
Given two selectors $s, s' \in S$, s' is said to be a *prefix* of s, written $\textit{prefix}(s', s)$, if s' is of the form $s'_1 \cdots s'_n$ with each $s'_i \in Gsel$, and s is of the form $s'_1 \cdots s'_n s_1 \cdots s_m$ with $m \geq 0$ and each $s_i \in Gsel$. When $m > 0$, s' is said to be a *proper prefix* of s, written $\textit{proprefix}(s', s)$.

Definition 5-2: Selector overlap and selector conflict
Two selectors s_i, s_j are said to *overlap*, written $\textit{overlap}(s_i, s_j)$, if
$\exists s' \, [\, \textit{prefix}(s', s_i) \wedge \textit{prefix}(s', s_j) \,]$
They are further said to *conflict*, written $\textit{conflict}(s_i, s_j)$, if $s' = s_i$ or $s' = s_j$.
It is trivially the case that $\forall s \in S \; [\; \textit{conflict}(s, s) \;]$.

If two selectors have a common prefix, then they specify access paths in an h-graph that require taking the value of one of more common nodes. Selecting *through* a node is a *read* data access, but when a node is the final selection of a selector it is potentially *writable*. A potential conflict, then, only arises if one of these common nodes is the final selection of one or both of the two selectors, *i.e.*, if $\textit{conflict}(s_i, s_j)$.

Recall that the definition of a procedure call in the previous chapter restricts the actual parameter selectors of a procedure call to selecting only atoms, not graphs. This prevents the creation of pointers that would allow access in one procedure to the data state in another procedure. In addition to this restriction, the analyses presented in this chapter assume that for each procedure in an HG software system model no selector is *aliased* with any other in the initial data state h-graph. The non-aliased assumption means that given two h-graph selectors s_i, s_j and an initial h-graph h,

$$[[s_i]](h) = [[s_j]](h) \;\; \textit{iff} \;\; s_i \equiv s_j .$$

The selectors designate different nodes unless they are textually identical.

For an initial data state free of aliases, the following lemmas and theorem ensure that execution of a procedure call will not introduce any aliases. The first lemma applies to a data state with no aliased nodes. It states that if an assignment is performed on a selected node, then every other selector designates the same node after the assignment as prior to it (except for those having the original selector as a proper prefix, which become undefined):

Lemma 5-1:

Given: $h \in \Gamma$, S an alias-free set of selectors on h, s and $s' \in S$ ($s \not\equiv s'$), $a \in \Delta$, and $h' = \alpha(h, s(h), a) \neq \bot$.

Then: either

(1) $\mathit{proprefix}(s, s') \wedge s'(h') = \bot$, or

(2) $s'(h) = s'(h')$.

Proof:

(1) Consider the case in which $\mathit{proprefix}(s, s')$. Then:

$s' \equiv ss''$ for some $s'' \in S$, by Definition 5-1, the meaning of a selector prefix;

$s'(h') = s''(V(s(h')))$ by Definition 3-8.a (the meaning of an h-graph selector);

$s'(h') = s''(a)$, by Definition 3-10.a (the value function produced by assignment); but since a is an atom, $s''(a) = \bot$;

$s'(h') = \bot$, by substitution.

(2) For the case of $\neg\mathit{proprefix}(s, s')$, let $s' = s'_1 s'_2 \cdots s'_k$. By Definition 3-8 (the meaning of an h-graph selector applied to an h-graph),

$$s'(h) = s'_k(V(s'_{k-1}(V(\cdots(V(s'_1(h)))\cdots)))).$$

Since the selectors are assumed non-aliased, and since s is assumed not a prefix of s', none of the nodes $s'_i(\cdots)$ has its value changed by the assignment creating h'. Thus for each i, $V(s'_i(\cdots)) = V'(s'_i(\cdots))$ where $h = \langle G, V, r \rangle$ and $h' = \langle G, V', r \rangle$. Then,

$$s'(h) = s'_k(V'(s'_{k-1}(V'(\cdots(V'(s'_1(h')))\cdots)))) = s'(h').$$

□

The next lemma states that no aliases can be created in a data state by an assignment to a node of that state, that is, s and s' are aliases in h' after an assignment only if s and s'

are aliases in h, prior to an assignment:

Lemma 5-2:
Given: h, h', and S as in Lemma 5-1, with $s, s' \in S$, and $s(h') = s'(h')$.
Then: $s(h) = s'(h)$.

Proof: By Lemma 5-1, we know that s cannot be a proper prefix of s', or the two cannot be aliases as is assumed. Therefore, by Lemma 5-1 we know that $s'(h) = s'(h')$. Combining this with the assumption gives $s(h') = s'(h)$. By the definition of the meaning of assignment, $s(h') = s(h)$, so combining again gives $s(h) = s'(h)$.

□

The following theorem attests that the action of a procedure call on a data state does not create aliases:

Theorem 5-1:
Given: an h-graph h, a set of selectors S which is alias-free on h, and a procedure call w such that $h' = w(h)$.
Then: S is alias-free on h'.

Proof: By contradiction. Assume that S is not alias-free in h'. By Definition 3-12.b, execution of w in h defines the sequence of assignments $\alpha(h_i, r_i(h_i), v_i) = h_{i+1}$, where $h = h_1$ and $h' = h_{n+1}$. By Lemma 5-2, the existence of aliases in $h' = h_{n+1}$ implies the existence of aliases in h_n. In general, the existence of aliases in h_{i+1} implies the existence of aliases in h_i. This in turn implies that S is not alias-free on $h_1 = h$, contradicting a premise. Therefore, the assumption that S is not alias-free on h' is false.

□

Theorem 5-1 says that we need only verify that the initial data state for a procedure is alias free in order to guarantee that all data states produced during execution are also alias-free and so suitable for the analyses we describe below. Most programming languages require alias-free initial data states. The problems of program verification in the presence of aliased h-graph structures are discussed in Wilson's thesis [64].

Two forms of read/write conflict can arise between two concurrent basic blocks. We offer a condition on the selectors in the blocks for each type of conflict. Given a basic block b_j and a set S_j of h-graph selectors appearing in the procedure calls of b_j, consider the following sets:

- S_j^w is the set of all selectors $s \in S_j$ that designate nodes which have their values assigned during the execution of b_j.
- $S_j^{wr} = S_j^{wr'} \cup S_j^{wr''}$, where $S_j^{wr'}$ is the set of all selectors $s' \in S_j$ that designate nodes which have their values assigned and then read one or more times with no intervening assignments during the execution of b_j, and where $S_j^{wr''}$ is the set of selectors $\{s'' \mid s' \in S_j^{wr'} \wedge \mathit{proprefix}(s'', s')\}$, the proper prefixes of the written-then-read selectors.
- $S_j^{rr} = S_j^{rr'} \cup S_j^{rr''}$, where $S_j^{rr'}$ is the set of all selectors $s' \in S_j$ that designate nodes which have their values read twice or more with no prior or intervening writes during the execution of b_j, and where $S_j^{rr''}$ is the set of selectors $\{s'' \mid s' \in S_j^{rr'} \wedge \mathit{proprefix}(s'', s')\}$, the proper prefixes of the read-then-read-again selectors.

To maintain the integrity of the computation being performed by b_j, a node selected by a selector s must have the same value each time it is accessed between writes or block boundaries. This may not happen if b_k is writing to s as well, depending on the relative order of the operations in the two blocks. A potential conflict is also created if b_k writes to the node designated by any prefix of s. This action could change from a graph to an atom the value of a node in the path by which s determines its selected node, causing s to become undefined.

Definition 5-3: Potential interference

We say that b_k *potentially interferes with* b_j, written *interfere*(b_k, b_j), if these conditions are met:

i. $S_k^w \cap S_j^{wr} \neq \varnothing$

b_k assigns to a node that either is one which b_j has assigned for later reference, or one which is in the access path to such a "scratchpad" node, or

ii. $S_k^w \cap S_j^{rr} \neq \varnothing$

b_k alters the value that b_j gets on two or more successive reads from a node m, or it assigns to any node in the access path to m.

We must also check for b_j potentially interfering with b_k to detect all possible conflicts between the two basic blocks.

Definition 5-4: Conflict between basic blocks
Two basic blocks b_j and b_k are in *conflict*, written *conflict*(b_j, b_k), if
interfere$(b_j, b_k) \vee$ *interfere*(b_k, b_j).

Note that no pair of basic blocks needs be checked in which the pair is split across two different procedure models (two different PFGs). The semantics of the software system model are that each procedure call executes a procedure with its own separate, local data area. Argument values are copied in at initiation and copied back at termination. The system is a hierarchy of independent storage areas, so read/write conflicts cannot occur except within a single procedure. Thus we consider a set B of basic blocks, all from the static program model of a single procedure.

From B, we then seek to identify all pairs of blocks that can possibly execute concurrently according to the structure of the timed Petri net *alone*. To determine this, no consideration is given to the data state and the effect it has on branching during actual model execution. We assume that if a path exists in the Petri net structure then there is some execution sequence that goes down that path. This simplification is justified by the observation that if conflicts are discovered and corrected in blocks that never actually execute concurrently (due to the dictates of the data state sequence), no harm is done to the computation. For a static program model $SP=\langle S, NP, B\rangle$ and a control flow model $CF=\langle TP, \beta, \sigma, \rho\rangle$, define the set of possibly concurrent blocks as

$$C_\phi = \{\langle b_i, b_j\rangle \mid b_i, b_j \in B,\ \beta(p)=b_i,\ \beta(p')=b_j,$$
$$\text{and } p,\ p' \text{ are marked in some reachable marking } \mu \text{ of } TP\}.$$

The concept of *reachable marking* is taken up in the next section. This restriction to a *reachable* marking is a weak condition, in that some block pairs may be identified as possibly concurrent that will never be actually concurrent when the data state sequence guides model execution. Formally, for a reachable marking μ of TP indicating that a pair $\langle b_i, b_j \rangle$ is possibly concurrent, there may not exist a computation $\zeta_0, \zeta_1, \cdots, \zeta', \cdots$ of CF in which $\zeta' = \langle \mu, h \rangle$. However, no block pairs that can achieve concurrent execution are overlooked by this definition of C_ϕ.

Definition 5-5: Concurrent conflict between basic blocks
Two basic blocks b_j and b_k are in *concurrent conflict*, written $\pi conflict(b_j, b_k)$, if

$$conflict(b_j, b_k) \wedge \langle b_j, b_k \rangle \in C_\phi.$$

Given these definitions, the problems we seek to solve may be phrased this way:

1. Given a PFG $\phi = \langle SP, CF \rangle$ where $SP = \langle S, NP, B \rangle$, construct the set C_ϕ, and then identify a set $C_\phi' \subset C_\phi$ such that $\pi conflict(b_j, b_k)$ for each $\langle b_j, b_k \rangle \in C_\phi'$.

2. Given a particular pair of basic blocks $\langle b_j, b_k \rangle \in C_\phi'$, construct a corresponding pair of basic block sets $\langle B_j, B_k \rangle$, and make appropriate changes to the control flow model, such that
   a. B_j, B_k model the same computation as b_j, b_k, and
   b. for each $b_j \in B_j$ and $b_k \in B_k$, $\neg \pi conflict(b_j, b_k)$.

5.2. Identifying concurrently executable basic blocks

We look first at methods of solution for the first of the two problems. In the PFG for a single procedure, only the pairs of blocks that can execute in parallel (according to the control flow alone) need be checked for interference. Given a PFG ϕ and its HG model components $\langle SP, CF \rangle$, consider the set C_ϕ of all pairs of basic blocks from SP that can possibly execute concurrently according to the execution structure defined by the Petri net in CF. We wish to determine if $\langle b_i, b_j \rangle \in C_\phi$ for any arbitrary pair of basic blocks b_i and $b_j \in B$.

Definition 5-6: CONBLOCKS: The Concurrent Blocks Problem
Given a PFG ϕ and two basic blocks b_i, b_j in ϕ, is $\langle b_i, b_j \rangle \in C_\phi$?

For a general parallel flow graph, one way to systematically identify concurrent conflicts is to assume that *all* pairs of blocks can possibly execute concurrently and check pairwise the entire set B. Since this solution could be excessively time consuming and could find textual conflicts that, due to sequential execution, are not truly problematical, we would like to find an alternative method.

We can completely solve CONBLOCKS by reducing it to another problem that has a known solution. Recall that we are considering here only Petri nets in which each place has a duration of one time unit, so that only the number of tokens at each place is significant state information. A marking μ, then, of a Petri net with places P, $|P|=n$, can be written as an n-tuple of non-negative integers, as in regular Petri net theory. We say that a marking μ' *covers* a marking μ, written $\mu' \geqslant \mu$, if $\forall p \in P\, [\; \mu'(p) \geqslant \mu(p)\;]$.

Definition 5-7: COVER: The coverability problem for general Petri nets
Given a Petri net N, an initial marking μ, and a second marking μ', is there a reachable third marking μ'' such that $\mu'' \geqslant \mu'$?

Theorem 5-2:
CONBLOCKS $\propto$ COVER.

Proof: We show that every instance of the concurrent blocks problem can be polynomially transformed into an instance of the coverability problem by describing the construction.

An instance of CONBLOCKS is a PFG $\phi=\langle SP, CF\rangle$, where $SP=\langle S, NP, B\rangle$, and two basic blocks $b_i, b_j \in B$. The component CF of ϕ contains a marked timed Petri net $TP_\phi=\langle N_\phi, \mu_{0\phi}, \tau_\phi\rangle$ where $N_\phi=\langle P_\phi, T_\phi, I_\phi, O_\phi\rangle$.

An instance of COVER is a Petri net structure $N=\langle P, T, I, O\rangle$, an initial marking μ, and another marking μ'.

Construct $N = N_\phi$, and construct $\mu = \mu_{0\phi}$.

To construct μ' we need the following observations:

i. A block b can possibly execute in a configuration $\zeta=\langle\mu, h\rangle$ of CF if, for some $p \in P_\phi$ where $\beta(p)=b$ and $t \in O(p)$, all $\mu_j \geqslant 1$ for $p_j \in I(t)$. This says that a block may execute if one of the places that maps to that block is an input to a fully enabled transition.

ii. The techniques used to build the Petri net from the PFG guarantee that each block is associated with a different Petri net component. The structure of all components is such that, for all $p \in P_\phi$,

- if $|O(p)|>1$, then $\forall t \in O(p)$, $I(t)=\{p\}$ (a P/nT component)
- if $|O(p)|=1$, then
  either $|I(O(p))|=1$ (a P/T component)
  or $\forall p_i \in I(O(p))$, $O(p_i)=O(p)$ (an nP/T component)

This says that a block b is either the image (under the basic block selection function β) of a single place that *alone* enables at least one transition, or it is the image of each of several places, all of which are inputs to a single transition.

Thus a block b may execute in a configuration ζ if *each* of the one or more places that are images of b under β contains one *or more* tokens. A marking which has these places marked for both of two basic blocks, then, would indicate a configuration in which both are executing.

Construct μ' so that

$$\mu'_k = \begin{cases} 1 & \text{if } p_k \in P^i_\phi \text{ or } p_k \in P^j_\phi \\ 0 & \text{otherwise} \end{cases}$$

where $P^i_\phi=\{p \in P_\phi \mid \beta(p)=b_i\}$, and $P^j_\phi=\{p \in P_\phi \mid \beta(p)=b_j\}$.

The Petri net construction techniques that generate the PFG ϕ guarantee that an instance of CONBLOCKS is answered "yes" if and only if that instance transformed to COVER is answered "yes".

□

Note that the theorem only holds for a static program and control flow model pair constructed from a parallel flow graph. Models constructed *ad hoc* may not have the regular structure relied upon in the proof. In particular, the basic block selection function β in an arbitrary model is not constrained to either map a single place into a block or to map a group of places, all inputs to a single transition, each into a single block.

Since COVER is at least as hard as CONBLOCKS, we can solve each instance of CONBLOCKS by solving the equivalent instance transformed to COVER. The solution to COVER is obtained by computing the (finite) *reachability tree* for the Petri net in the problem instance, then searching it for a state satisfying the coverability conditions.

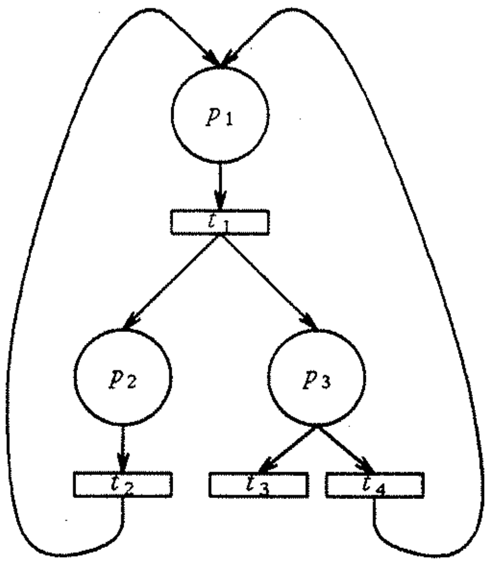


Figure 5.1 Simple Petri net

5.2.1. The Petri net reachability tree

The reachability tree for a normal Petri net with an initial marking is a graphical representation of the *reachability set*, that is the set of all markings (states) of the Petri net that can be reached from the initial marking (initial state) by successive applications of the execution rule. Each node in the tree represents a particular state, and an arc leaving a node is labeled with the transition that produces the state that the following node represents. From any given node (current state) in the tree, there are as many arcs to follow to a next node (next state) as there are elements in the set of enabled transitions for the current state.

For many Petri nets, the reachability tree is infinite. This occurs when at least one transition sequence produced by execution of a Petri net is infinite. Consider the Petri net shown in Figure 5.1. Its reachability tree is pictured in Figure 5.2, and can be seen to

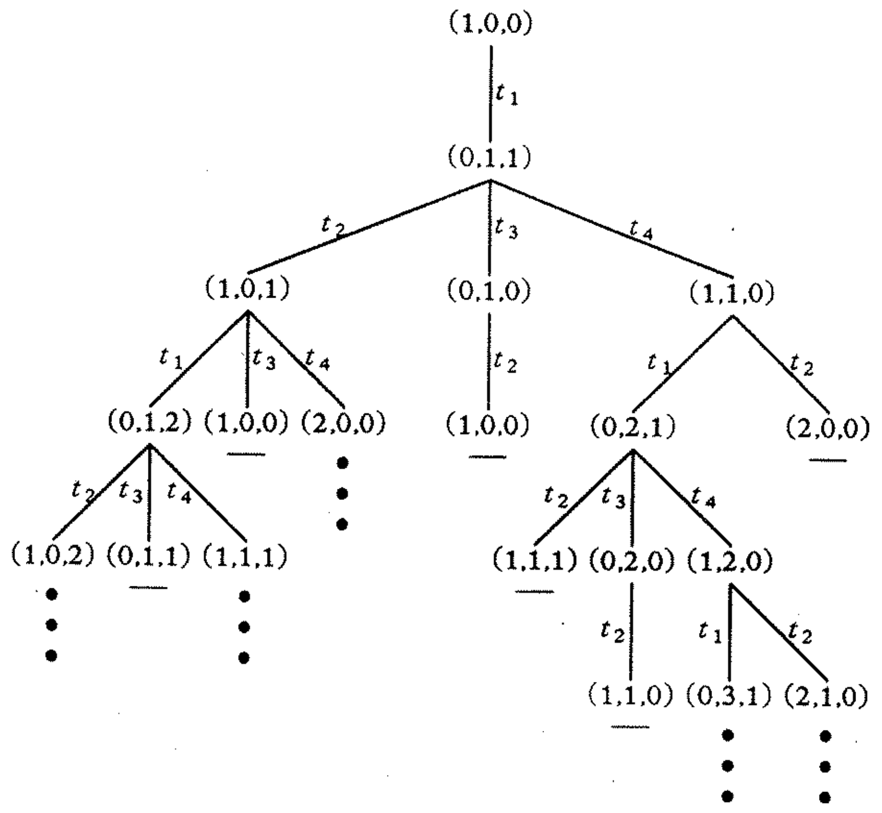


Figure 5.2 Reachability tree for net in Fig. 5.1

contain several infinite paths, represented by the elided nodes. The underscored nodes represent states that are duplicates of states found elsewhere in the tree[1]. Notice that as certain sequences of transitions are repeated, the number of tokens is increasing regularly in some places of the markings on the infinite paths. A finite form of the reachability tree can be produced in which collapsed single states are created from these groups of states that are equivalent except for an arbitrarily large number of tokens in some of the places. The special symbol ω is used in the distinguished states to represent the arbitrarily large number of tokens in a place. If the state sequence from some state μ in the tree back to

[1]These duplicate nodes mean that we are actually representing a reachability graph with cycles, but the designation of *tree* is retained to refer to the structure as constructed here.

the root contains a state $\mu' \neq \mu$ such that $\forall i \left[\mu(p_i) \geqslant \mu'(p_i) \right]$, then each $\mu_i > \mu'_i$ can be replaced by an ω. The algebra of ω is given by

$$\omega \pm c = \omega$$

$$\omega \geqslant c$$

$$\omega \geqslant \omega$$

where c is any arbitrary constant. The resulting tree has leaf nodes which are either terminal states of the net, *i.e.*, no transitions are enabled, or states which are duplicates of ones found elsewhere in the tree. Peterson gives an algorithm and a proof of termination in [51] for constructing a finite representation of the reachability tree for a Petri net which is executed under the normal, one-at-a-time transition firing rule. Figure 5.3 shows a finite form of the reachability tree from Figure 5.2.

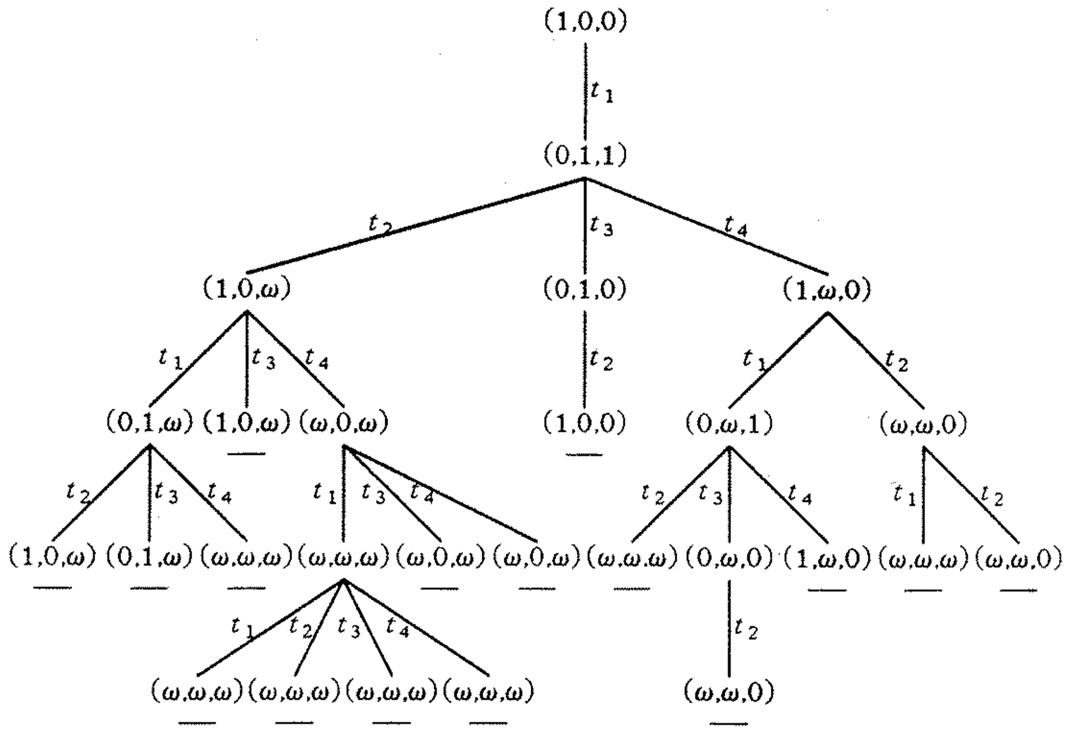

(1,0,0)
t_1
(0,1,1)
t_2 t_3 t_4
(1,0,ω) (0,1,0) (1,ω,0)
t_1 t_3 t_4 t_2 t_1 t_2
(0,1,ω) (1,0,ω) (ω,0,ω) (1,0,0) (0,ω,1) (ω,ω,0)
t_2 t_3 t_4 t_1 t_3 t_4 t_2 t_3 t_4 t_1 t_2
(1,0,ω) (0,1,ω) (ω,ω,ω) (ω,ω,ω) (ω,0,ω) (ω,0,ω) (ω,ω,ω) (0,ω,0) (1,ω,0) (ω,ω,ω) (ω,ω,0)
t_1 t_2 t_3 t_4 t_2
(ω,ω,ω) (ω,ω,ω) (ω,ω,ω) (ω,ω,ω) (ω,ω,0)

Figure 5.3 Finite form of the reachability tree in Fig. 5.2

Some information is lost in the transformation from a full reachability tree to a finite form; specifically, the lengths of the transition sequences between specific instances of the ω states cannot be determined. The finite form of a reachability tree is not unique in structure; for instance, generating the tree breadth-first for a particular net gives a different placement of the duplicate nodes than if the same Petri net were used with the tree nodes generated depth-first. The different representations, however, are collectively unique in that one contains the same useful information as another about the generating Petri net. The finite tree is sufficient to solve some analysis problems, most notably (for our purposes) the coverability problem. Given the finite reachability tree for a Petri net, an instance of COVER can be solved simply by searching all the nodes of the tree to find a marking which covers the target marking μ'. In the remaining discussion we shall refer to a finite form of the reachability tree for a Petri net as simply *the* reachability tree, unless otherwise indicated.

5.2.2. The concurrent reachability tree

We cannot employ the reachability tree directly as Peterson describes it. Since the tree is a representation of all possible state sequences, its structure is dependent on the execution rule. Our concurrent firing rule excludes some of the states that can be obtained under the normal rule, so we must alter the tree construction to reflect this property.

An example will illustrate this point. Consider again the timed Petri net shown in Figure 5.1, and its reachability tree under the normal firing rule in Figure 5.2. Under the concurrent execution rule we have adopted for Petri nets, the reachability tree generated is the one in Figure 5.4. Though the tree is infinite as before, notice the lack of state proliferation caused by delaying enabled transitions from firing under the normal execution rule. We can see that the reachability tree for a normal Petri net contains some branches (and so introduces some computational complexity) which, under our interpretation, are

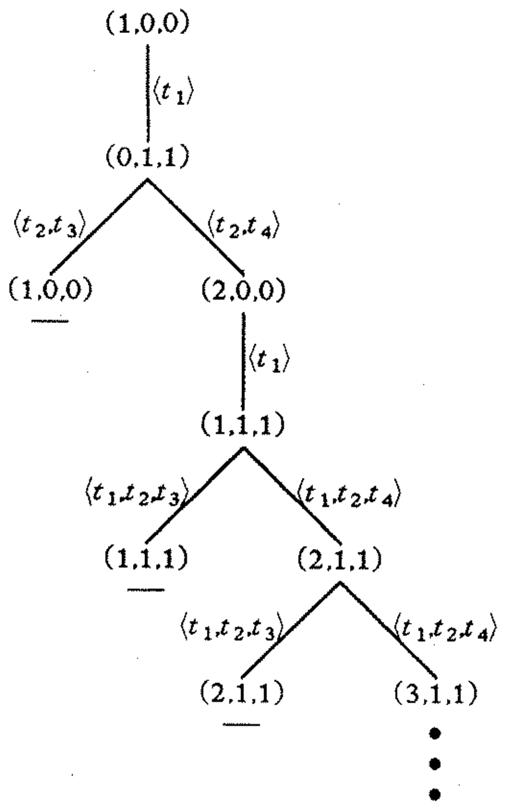


Figure 5.4 Concurrent reachability tree for net in Fig. 5.1

not due to the algorithm, but rather to the execution host. We eliminate those branches in the concurrent reachability tree, and so deal only with alternatives dictated by the algorithm. The effect is that the concurrent reachability tree of a Petri net under the maximal execution rule represents the maximum possible concurrency--the ideal against which implemented systems can be compared.

Now consider creating ω states in this concurrent reachability tree. If we apply the algorithm Peterson describes for the sequential execution rule, the tree pictured in Figure 5.5 is obtained. The state marked * in this tree implies that concurrent execution should be able to generate a state μ such that $\mu(p_1) \geqslant 3$, $\mu(p_2)=0$, $\mu(p_3)=0$. Looking at the full tree in Figure 5.4 we see that such a state is in fact not obtainable. Thus the algorithm

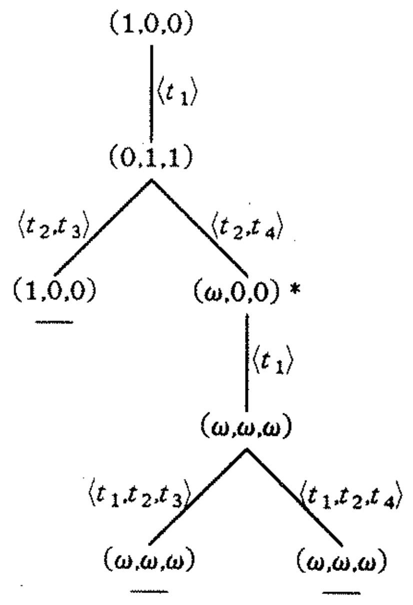


Figure 5.5 Incorrect finite representation of tree in Fig. 5.4

for creating ω states under the sequential firing rule is not sufficient for the concurrent firing rule.

The reason the symbol ω can be freely inserted into a net marking when constructing a sequential reachability tree is because the one-at-a-time firing rule allows exact reproduction of the transition firing sequence that leads from a marking μ to another one μ' where $\mu' \geqslant \mu$. Hence the notion of "pumping" a place or places with tokens is an achievable execution phenomenon. Under a concurrent firing rule, pumping may not be possible. Transitions may be enabled in μ' that were not enabled in μ, and their firing cannot be denied (as they may under the normal firing rule) in order to exactly repeat the sequence of transition set firings that led to μ' from μ. Thus the conditions for inserting ω markings must be strengthened in order to build a finite representation of the reachability tree under the concurrent firing rule.

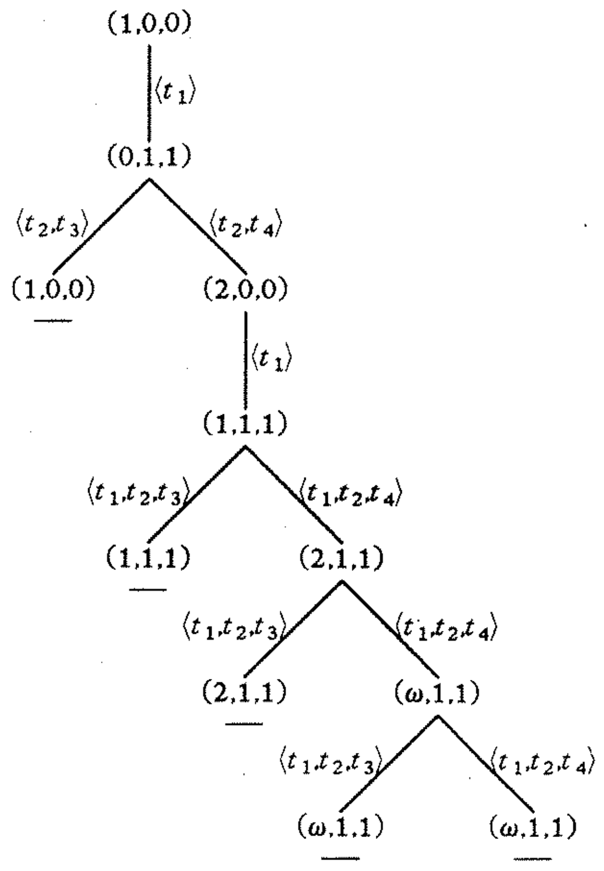


Figure 5.6 Correct finite form of the reachability tree in Fig. 5.4

Given a current marking μ, consider another marking μ' on the path back to the root from μ. Simply checking to see if $\mu \geqslant \mu'$ is not enough, for μ may create some enabled transitions that are not enabled in μ'. Firing the net, then, in state μ may, through these extra transitions, have an eventual effect other than to pump tokens into some places. Extra enabled transitions can occur in μ when a place p_i goes from having no tokens to having one (or more) tokens. This means that if $\mu'(p_i)=0$ and $\mu(p_i) \geqslant 1$ we cannot expect that the same transitions will necessarily fire from μ as from μ'. Furthermore, if $\mu'(p_i)=1$ and $\mu(p_i) \geqslant 2$, then we still cannot add the ω symbol for p_i. This situation implies that in the state following μ there may be transitions enabled by the token left in

p_i that could not have been enabled in the state following μ', where the number of tokens in p_i will have dropped to 0. We need to consider not only the firing possibilities in the current state μ, but the ones that may result in the states immediately following μ as well. In terms of being able to repeat the exact transition-set firing sequence leading from μ' to μ, a steady state at place p_i is only possible if p_i contains 2 or more tokens in the earlier state μ'. Then the token count cannot drop to zero in the state following μ', as it also cannot in the state following μ.

Pumping tokens, then, into a place p_i is only possible if the current marking μ is across-the-board nondecreasing with respect to some previous marking μ', and p_i has at least 2 tokens in μ'. When constructing the concurrent reachability tree, then, ω symbols can be added to a state μ at place p_i if

$$\exists \mu' \overset{*}{\Longrightarrow} \mu : \left(\forall p_k \left[\mu(p_k)^\nu \geqslant \mu'(p_k)^\nu \right] \wedge \ \mu(p_i)^\nu > \mu'(p_i)^\nu \ \wedge \ \mu'(p_i)^\nu > 1 \right)$$

where $\overset{*}{\Longrightarrow}$ means "precedes in some state sequence from the initial state." Note that the stronger conditions employed to construct the concurrent reachability tree will also produce an acceptable tree for a sequential execution rule. Because of the delay in detecting pumping of a place until two or more tokens are present, the resulting tree may simply have more nodes in it than one constructed via Peterson's rules. Figure 5.6 illustrates the concurrent reachability tree obtained from the strengthened algorithm for the example Petri net in Figure 5.1.

One final alteration we apply to Peterson's tree-generating algorithm is that the nodes are required to be generated in a breadth-first fashion. This feature is employed later in arguing about the distance that a duplicating node is from the root of a tree in relation to the node it duplicates. Algorithm 5.1 presents a formal summary of the altered concurrent reachability tree construction procedure.

As with Peterson's algorithm, the algorithm to construct the concurrent reachability tree terminates. The proof follows the main lines of Peterson's previously mentioned

```
algorithm build_reachability_tree
--
-- The reachability tree is constructed in much the same way as described in Peterson,
-- p. 95, with the addition of stricter conditions for inserting $\omega$ symbols in a net state
-- (tree node). The extra conditions are a reflection of the net behavior under the
-- concurrent transition firing rule used in net execution.
--
-- This version assumes no times on the places of the Petri net, that is, a marking $\mu$ of a
-- net is defined solely in terms of the number of tokens at each place.
--
-- The choice of which frontier node to process next must be made in a manner that
-- generates the tree breadth-first. This organization is assumed in the arguments
-- about path length and duplicate nodes.
--
root $\leftarrow \mu_0$
nodetype ( root ) $\leftarrow$ frontier
while $\exists$ x s.t. nodetype ( x ) = frontier
    if $\exists$ y s.t. nodetype ( y ) $\neq$ frontier and $\mu_x = \mu_y$
        nodetype ( x ) $\leftarrow$ duplicate
    else if $T_e = \emptyset$ for $\mu_x$
        nodetype ( x ) $\leftarrow$ terminal
    else -- x is not duplicate, and there are transitions enabled
        nodetype ( x ) $\leftarrow$ interior
        for each maximal firable set $T_e' \subset T_e$
            create a new node z
            nodetype ( z ) $\leftarrow$ frontier
            $\mu_z \leftarrow$ execute $T_e'$ in $\mu_x$
            for each place $p_i$
                if $\exists$ y on the path from root to x
                s.t. $\mu_z \geqslant \mu_y$ and $\mu_z(p_i) > \mu_y(p_i)$ and $\mu_y(p_i) > 1$
                    $\mu_z(p_i) \leftarrow \omega$
end build_reachability_tree
```

Algorithm 5.1 Constructing a Petri net concurrent reachability tree

proof, with an additional argument about markings in which some places have two or more tokens. We include two of Peterson's supporting lemmas for ease of argument, and refer the reader to his text [51] for their proofs.

Lemma 5-3:
In any infinite directed tree in which each node has only a finite number of directed successors, there is an infinite path leading from the root.

Proof: Peterson [51], page 97 top.

□

The concurrent reachability tree has at each node a bounded number of direct successors. Since a successor is obtained by firing some subset of the enabled transitions, in the case of the most general execution rule the bound is given by $2^{|T|}$.

Lemma 5-4:
Every infinite sequence of n-vectors over the nonnegative integers plus the ω symbol contains an infinite nondecreasing subsequence.

Proof: Peterson [51], page 98 top.

□

With these lemmas we can prove that the construction algorithm terminates.

Theorem 5-3:
The concurrent reachability tree of a Petri net is finite.

Proof: By contradiction.

Postulate an infinite concurrent reachability tree. By lemma 5-3, then, there exists an infinite sequence of markings (n-vectors) $\mu_0, \mu^1, \mu^2, \cdots$ from the root of the tree. By lemma 5-4 this sequence contains an infinite nondecreasing subsequence $\mu^{i_0} \leqslant \mu^{i_1} \leqslant \mu^{i_2} \leqslant \cdots$. In this subsequence there cannot exist two markings such that $\mu^{i_j} = \mu^{i_k}$, since by the construction algorithm one would be a duplicate of the other and the sequence would terminate. Thus the infinite subsequence is strictly increasing, $\mu^{i_0} < \mu^{i_1} < \cdots$.

Consider all n-vectors over $\{0,1,2\}$. There are 3^n (a finite number) unique n-vectors over this set, so eventually we must find two markings in the sequence such that $\mu^{i_j} < \mu^{i_k}$ and for a place p_l, $\mu^{i_k}(p_l) \geqslant 3$ when $\mu^{i_j}(p_l) \geqslant 2$. By construction, $\mu^{i_k}(p_l)$ becomes ω, and the argument now is applied to vectors of length $n-1$. Eventually a state μ^{i_m} is encountered in which $\forall j [\mu^{i_m}(p_j) = \omega]$. There can be no $\mu^{i_{m+1}} > \mu^{i_m}$, so the assumption of an infinite concurrent reachability tree is false.

□

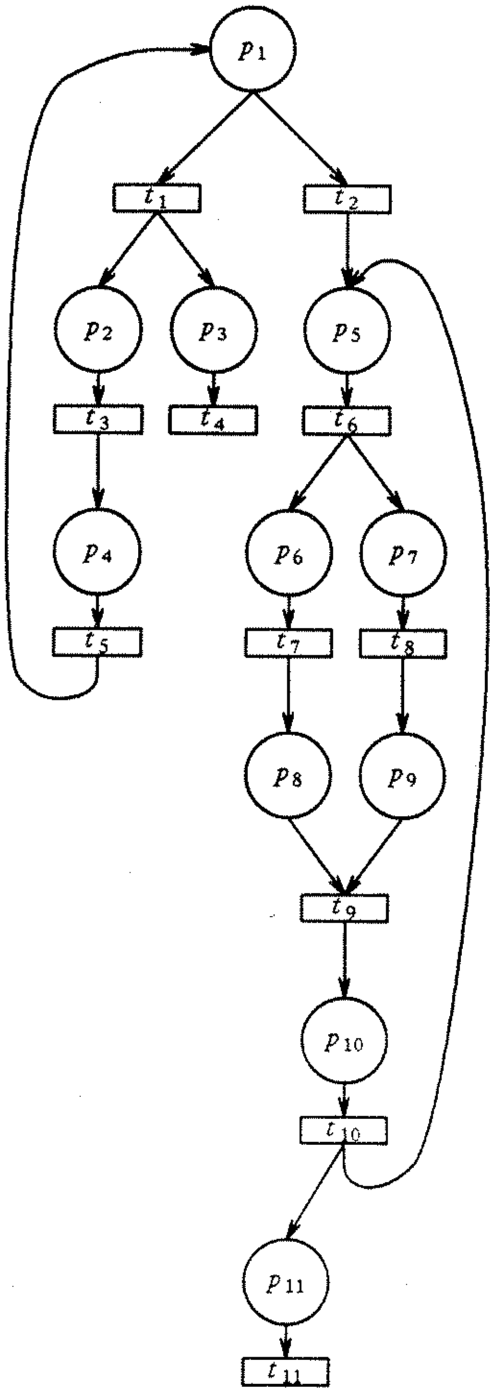
5.2.3. The concurrency matrix

Given a PFG ϕ representing the static program and control flow models of a software system, the CONBLOCKS problem for each pair of basic blocks can be completely solved as the concurrent reachability tree for the Petri net is constructed. A square boolean matrix MC_ϕ, termed the *concurrency matrix*, is generated that is of size $|B|$, the meaning of which being that if $MC_\phi[i,j]$ is **true** (and symmetrically $MC_\phi[j,i]$), then $\langle b_i, b_j \rangle \in C_\phi$.

The elements of the matrix are filled in by first starting with $\forall 1 \leqslant i,j \leqslant |B| \left(MC_\phi[i,j] = \textbf{false} \right)$. As each state μ is generated during construction of the reachability tree for the Petri net, the marking is scanned for marked places, *i.e.*, places for which $\mu(p) \geqslant 1$. Each distinct pair of such places causes an element of MC_ϕ to be set to **true** . Formally, this is stated

$$\forall 1 \leqslant i,j \leqslant |B| \left(p_i \geqslant 1 \wedge p_j \geqslant 1 \wedge i \neq j \implies b_k = \beta(p_i) \wedge b_l = \beta(p_j) \wedge MC_\phi[k,l] = \textbf{true} \right).$$

Figure 5.7 illustrates the difference the firing rule makes in determining the size of C_ϕ. A "■" in the diagram indicates the potential concurrency of the pair of blocks given by the row and column it occupies; a blank indicates no potential concurrency, and a "O" marks the diagonal elements as ignored. In this example, the basic blocks represented in the matrices are associated with places having the same subscript, *i.e.*, $\beta(p_i)=b_i$. The two concurrency matrices describe the potential behavior of the Petri net pictured when executed under each of the respective transition firing rules. A comparison of the shaded cells in each shows the exclusion under the concurrent rule of blocks that are potentially parallel under the sequential rule only as a result of unnecessary delays (*e.g.*, due to the unavailability of processors).



Sequential Firing Rule

11			■		■	■	■	■	■	■	○
10			■							○	■
9			■			■		■	○		■
8			■				■	○	■		■
7			■			■	○	■			■
6			■			○	■		■		■
5			■		○						■
4			■	○							
3	■	■	○	■	■	■	■	■	■	■	■
2		○	■								
1	○		■								
	1	2	3	4	5	6	7	8	9	10	11

Concurrent Firing Rule

11					■						○
10										○	
9								■	○		
8								○	■		
7						■	○				
6						○	■				
5					○						■
4				○							
3		■	○								
2		○	■								
1	○										
	1	2	3	4	5	6	7	8	9	10	11

Figure 5.7 The concurrency matrix

5.2.4. Identifying conflicts among members of C_ϕ

Given that C_ϕ identifies basic block pairs that are possibly parallel in their execution, for each block pair $\langle b_j, b_k \rangle \in C_\phi$ the conditions for determining *conflict*(b_j, b_k) are applied. Recall that Definition 5-4 states these conditions as one block *potentially interferes* with the other, and that Definition 5-3 identifies b_k as potentially interfering with b_j if either

i. $S_k^w \cap S_j^{wr} \neq \emptyset$, or

ii. $S_k^w \cap S_j^{rr} \neq \emptyset$

for the sets of selectors S_k^w, S_j^{wr}, and S_j^{rr}. By application of these conditions to all members of C_ϕ, a set $C_\phi' \subset C_\phi$ is created which is used in the conflict resolution procedure described in the following sections.

5.3. Concurrent conflict resolution

We now look at solving the second of the two problems mentioned earlier in this chapter. After following the procedure described above, a model may be found to contain several pairs of concurrently conflicting basic blocks. These conflicts must be resolved in order for the model to be considered to represent a valid concurrent computation. If a pair is found to conflict, each is subdivided, if possible, in such a way as to allow the greatest possible overlapping of execution and the minimum amount of enforced sequential execution. Given that the two blocks in concurrent conflict are b_j , b_k $\in C_\phi'$, there are three main steps in this resolution:

1. Identify critical regions in b_j and b_k that are associated with each contested selector;
2. Partition b_j and b_k into a set of smaller basic blocks based on the critical regions;
3. Generate new portions of the Petri net model of the PFG to reflect the new basic block structure and to force the isolated critical regions to execute in mutual exclusion.

This solution provides enforced sequential execution of the conflicting sections of code, but the sequencing only goes into effect if the potential concurrency turns out to be actual concurrency during execution. The conflicting procedure calls may, due to the execution times of their respective preceding blocks, never actually attempt to run in parallel. The concurrency identification procedure we have discussed simply specifies blocks that, according to the structure of the PFG alone, could perhaps run concurrently. We now look at these steps in more detail.

Step 1.

Identification of the critical regions in the two basic blocks can be accomplished by examining more closely the potential interference conditions (i) and (ii), presented in Definition 5-3 and restated above. Each condition is an intersection of two sets: a set of selectors from the first block for nodes that are assigned values during execution; a set of selectors from the second block, the union of selectors for nodes whose values are required in execution and the proper prefixes of those selectors. If a condition is violated, then the intersection is non-null. The selectors that are members of an intersection, then, designate the nodes involved in a read/write conflict of the type represented by the particular condition. Given these candidate locations, we search the argument lists of the procedure calls from each basic block to determine which ones use the contested storage. The critical regions are constructed for one contested selector at a time.

Given a location s, the write-read conflict identified by condition (1) produces one or more critical regions in each basic block. The ones in b_j each extend across all procedure calls from a write to s up to and including the last read of s before another write to s (or before the end of the block). The ones in b_k are each a single procedure call which writes to location s.

The read-read conflict identified by condition (2) above is slightly different, so correspondingly different critical regions are created when it is violated. Only one region is created in b_j. It extends across all the procedure calls from the first read of s to the last read of s before the first write to s or the end of the block. As before, the regions created in b_k are each a single procedure call which writes to location s.

Iterate this process over all the selectors in the conflict intersections, and all conflicting critical regions that violate the corresponding condition will have been identified, each associated with a variable. Do this for each of conditions (i) and (ii), and all critical regions involved in b_k interfering with b_j will have been found. Repeat the entire procedure for b_j interfering with b_k to get all conflicting critical regions from b_j and b_k.

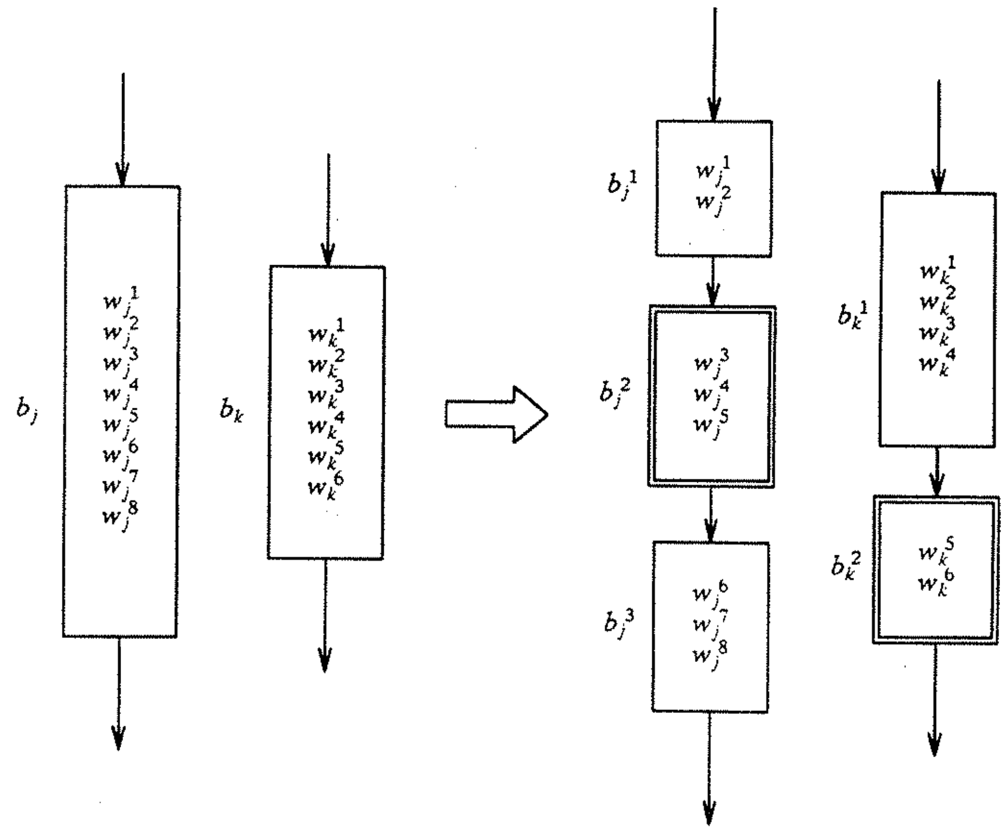

b_j
w_j^1 w_j^2 w_j^3 w_j^4 w_j^5 w_j^6 w_j^7 w_j^8
b_k
w_k^1 w_k^2 w_k^3 w_k^4 w_k^5 w_k^6
b_j^1
w_j^1 w_j^2
b_j^2
w_j^3 w_j^4 w_j^5
b_j^3
w_j^6 w_j^7 w_j^8
b_k^1
w_k^1 w_k^2 w_k^3 w_k^4
b_k^2
w_k^5 w_k^6

Figure 5.8 Isolation of conflicting critical regions

Step 2.

Once the critical regions of interest have been found, we must coalesce overlapping ones and isolate each of the resulting larger critical regions in its own basic block. Since each procedure call may use several selectors, it may be located in several critical regions, each dealing with a different storage location. For each of b_j and b_k, the overlaps are identified and the individual critical regions are merged by taking the union of the procedure calls in each as the new critical region. The larger region now involves several variables rather than one. The result is that each of the original basic blocks is partitioned into one or more non-overlapping sections of procedure calls.

Each critical region so identified is then represented in the static program model as a separate new basic block. An example will best explain this procedure. For b_j consisting of eight procedure calls in sequence and b_k consisting of six procedure calls, consider a partition in which the critical regions are:

$$\langle w_j^1, w_j^2 \rangle; \ \langle w_j^3, w_j^4, w_j^5 \rangle; \ \langle w_j^6, w_j^7, w_j^8 \rangle$$

and

$$\langle w_k^1, w_k^2, w_k^3, w_k^4 \rangle; \ \langle w_k^5, w_k^6 \rangle.$$

Replace b_j by b_j^1 containing procedure calls w_j^1 and w_j^2 from b_j, followed by b_j^2 containing w_j^3, w_j^4, and w_j^5, followed by b_j^3 containing w_j^6, w_j^7, and w_j^8. These three new blocks are connected in sequence so that the order of execution of the component procedure calls remains as in b_j. Replace in the same manner b_k by b_k^1 containing the first four procedure calls from b_k, followed by b_k^2 containing the last two calls. This construction is illustrated by Figure 5.8.

Step 3.

The final step involves regeneration of a part of the Petri net contained in the control flow model of the computation. Since we have increased the number of basic blocks

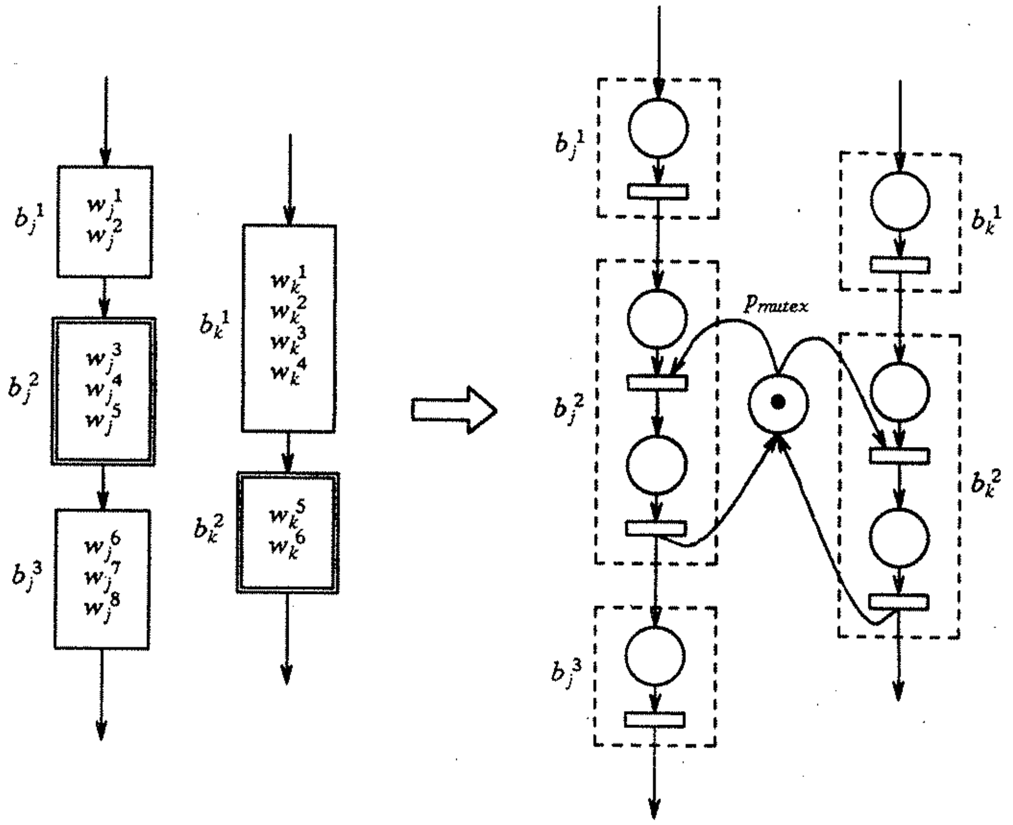


Figure 5.9 Petri net representation of conflicting critical regions

in the static program model, we must add corresponding P/T components to the Petri net. However, we must also add some synchronization structures to ensure that the new basic blocks which conflict never are allowed to execute simultaneously. We wish them all to execute in pairwise mutual exclusion, as in a critical region. The Petri net component, then, that represents a basic block in potential conflict is somewhat different from the simple P/T component that we introduced in the previous chapter.

Each basic block representing a critical region produces two P/T pairs in the Petri net; for reference designate them p_{block}/t_{block} and p_{sync}/t_{sync}. As in the earlier P/T pairs, an arc is directed from each place to its corresponding transition. In addition, an arc runs from t_{sync} to p_{block} to join the two pairs. The basic block representing the critical region is associated by the altered function β with p_{block}; a newly synthesized block, either null or containing instructions to perform synchronization, is associated with p_{sync}. The other new basic blocks resulting from the fragmentation of the two original ones are each represented by a single P/T net component as before. The Petri net components are connected together by arcs between them just as the arcs in the PFG connect the basic blocks they represent.

Pairs of the new basic blocks must now be mutually excluded. The pairs of interest are determined by the conflicts that were originally identified. For each original conflict, representing a read-read or write-read interference on one variable common to both b_j and b_k, each of the new basic blocks which contains a corresponding critical region of b_j must be mutually excluded from each of the new blocks containing a corresponding critical region from b_k. Any pair of blocks is excluded at most once, no matter how many individual variable interferences it may represent.

For each pair of blocks that must be mutually excluded, a single extra place, p_{mutex}, is created to serve as the mutual exclusion sentinel over the conflicting blocks. An arc is directed from p_{mutex} to the t_{sync} for each of the two conflicting blocks, and another arc is directed from each t_{block} back to p_{mutex}. A single token is placed in p_{mutex} to become part of the initial configuration of the Petri net model. A newly synthesized basic block, either null or containing instructions for performing synchronization, is associated by β with p_{mutex}. Figure 5.9 shows this conversion to Petri net components.

The timing of the original basic blocks must be altered for the new structure. The newly created basic blocks must be assigned execution times that are some proportion of the block from which they were made. These figures can be obtained from the execution times of the procedure calls contained in them. For a block that does not contain the conflicting procedure calls, the total execution time is ascribed to the place in its corresponding P/T Petri net component. For a basic block that contains one of the conflicting procedure calls, the place in the second P/T pair, that is p_{block}, is ascribed the block execution time; the other places, p_{sync} and p_{mutex}, are given times determined by the instructions necessary to perform the synchronization.

5.3.1. Extension to sets of conflicting blocks

The analysis as presented applies to a single pair of basic blocks. If a basic block is in concurrent conflict with several other blocks, the resolution procedure must be extended to produce a correctly synchronized model. One obvious method is to avoid the added complications and treat each basic block involved in a multiple conflict as a single critical region, and mutually exclude all of it rather than trying to subdivide it into new blocks. Though this method is easy to implement, it forces sequential execution on sections of the blocks that may safely proceed in parallel.

To avoid unnecessary sequencing, a more complicated procedure is required. When a basic block b_j is in conflict with more than one other block, several of the elements in row j of MC_ϕ will be **true** . For each row j of MC_ϕ, identify the critical regions (as in step 1 above) for each pair $\langle b_j, b_k \rangle$, where $MC_\phi[j,k] =$ **true** . Do not coalesce regions at each step, but collect them into a set of critical regions defined by *all* the blocks in conflict with b_j. After all critical regions have been found, then coalesce those that overlap, and keep with each critical region a list of the selectors involved in a conflict in that region. Apply step 2 above to the final set of critical regions to get a partition of new basic blocks in the static program model. The corresponding pair of P/T components for each new block can be added to the Petri net as well, but no mutual exclusion places and arcs are added at this point.

After all rows of MC_ϕ have been processed, go back through the matrix row-by-row a second time to add mutual exclusion to the Petri net in the control flow model. Within each row j, consider each pair $\langle b_j, b_k \rangle$ in conflict. Consider all pairs of new blocks, one from each partition formed from b_j and b_k. Intersect the critical selector lists, and if the intersection is not empty add a mutual exclusion place, token, and arcs to the appropriate P/T pairs, as described in step 3 above. Repeat this for each conflicting b_k in row j, and repeat the entire procedure for each row j in MC_{ϕ}.

5.4. An extended example (continued)

Continuing the example started in Chapter 4, we consider the model for the main procedure *quad*. Let the timing function τ map each place p_i in the Petri net for *quad* into the integer 1. The concurrent reachability tree can then be constructed for the this net, and is shown in Figure 5.10. Because it has 62 nodes, the tree is presented in a textual form rather than a graphical form.

The concurrency matrix derived from this reachability tree is shown in Figure 5.11. As used earlier in Figure 5.7, a "■" in the diagram indicates the potential concurrency of the pair of blocks given by the row and column it occupies; a blank indicates no potential concurrency, and a "O" marks the diagonal elements as ignored. For this example there are $\frac{11^2-11}{2}=55$ unique pairs of basic blocks. Of these pairs, 31 are potentially concurrent and 24 are not. Notice that in the state represented by node 26 in the reachability tree above, places 3, 8, and 10 are marked. As the concurrency matrix is being constructed, when state 26 is scanned the elements (3,8), (3,10), (8,10), and symmetrically (8,3), (10,3), (10,8), are all marked (if they are not marked already) to indicate that the basic blocks corresponding to these pairs of places are active in state 26.

From the elements marked in the concurrency matrix, pairs of blocks that share data in a possibly compromising way are identified. Because of the structure of the data state, a pair need only be considered if either block accesses a selector with the prefix */shared*. All pairs involving the blocks corresponding to places 1, 3, 6, 8, 9, 10, and 11 are eliminated from consideration by this condition. Only the pairs (2,5), (2,7), (4,5),

```
s1:(10000000000) is interior
  --1--> s2
s2:(01001010000) is interior
  --2-6-9--> s3
s3:(00100101000) is interior
  --4-8-11--> s4
  --4-8-10--> s9
  --4-7-11--> s10
  --4-7-10--> s19
  --3-8-11--> s20
  --3-8-10--> s37
  --3-7-11--> s38
  --3-7-10--> s62
s4:(00000010110) is interior
  --9--> s5
s5:(00000001110) is interior
  --11--> s6
  --10--> s7
s6:(00000010110) duplicates s4
s7:(00000000111) is interior
  --12--> s8
s8:(00000000000) is terminal
s9:(00000000111) duplicates s7
s10:(00001010100) is interior
  --6-9--> s11
s11:(00000101100) is interior
  --8-11--> s12
  --8-10--> s13
  --7-11--> s14
  --7-10--> s15
s12:(00000010110) duplicates s4
s13:(00000000111) duplicates s7
s14:(00001010100) duplicates s10
s15:(00001000101) is interior
  --6--> s16
s16:(00000100101) is interior
  --8--> s17
  --7--> s18
s17:(00000000111) duplicates s7
s18:(00001000101) duplicates s15
s19:(00001000101) duplicates s15
s20:(00010010010) is interior
  --5-9--> s21
s21:(01000001010) is interior
  --2-11--> s22
  --2-10--> s36
s22:(00100010010) is interior
  --4-9--> s23
  --3-9--> s24
s23:(00000001110) duplicates s5
s24:(00010001010) is interior
  --5-11--> s25
  --5-10--> s35
s25:(01000010010) is interior
  --2-9--> s26
s26:(00100001010) is interior
  --4-11--> s27
  --4-10--> s28
  --3-11--> s29
  --3-10--> s30
s27:(00000010110) duplicates s4
s28:(00000000111) duplicates s7
s29:(00010010010) duplicates s20
s30:(00010000011) is interior
  --5--> s31
s31:(01000000011) is interior
  --2--> s32
s32:(00100000011) is interior
  --4--> s33
  --3--> s34
s33:(00000000111) duplicates s7
s34:(00010000011) duplicates s30
s35:(01000000011) duplicates s31
s36:(00100000011) duplicates s32
s37:(00010000011) duplicates s30
s38:(00011010000) is interior
  --5-6-9--> s39
s39:(01000101000) is interior
  --2-8-11--> s40
  --2-8-10--> s41
  --2-7-11--> s42
  --2-7-10--> s61
s40:(00100010010) duplicates s22
s41:(00100000011) duplicates s32
s42:(00101010000) is interior
  --4-6-9--> s43
  --3-6-9--> s44
s43:(00000101100) duplicates s11
s44:(00010101000) is interior
  --5-8-11--> s45
  --5-8-10--> s46
  --5-7-11--> s47
  --5-7-10--> s48
s45:(01000010010) duplicates s25
s46:(01000000011) duplicates s31
s47:(01001010000) duplicates s2
s48:(01001000001) is interior
  --2-6--> s49
s49:(00100100001) is interior
  --4-8--> s50
  --4-7--> s51
  --3-8--> s52
  --3-7--> s53
s50:(00000000111) duplicates s7
s51:(00001000101) duplicates s15
s52:(00010000011) duplicates s30
s53:(00011000001) is interior
  --5-6--> s54
s54:(01000100001) is interior
  --2-8--> s55
  --2-7--> s56
s55:(00100000011) duplicates s32
s56:(00101000001) is interior
  --4-6--> s57
  --3-6--> s58
s57:(00000100101) duplicates s16
s58:(00010100001) is interior
  --5-8--> s59
  --5-7--> s60
s59:(01000000011) duplicates s31
s60:(01001000001) duplicates s48
s61:(00101000001) duplicates s56
s62:(00011000001) duplicates s53
```

Numbers on an arrow indicate transitions that fire to cause a change from the state (marking) above the arrow to the state number following the arrow. Multiple arrows below a state indicate alternative sets of transitions that may fire from that state.

Figure 5.10 Concurrent reachability tree for extended example

11		■	■	■	■	■			■	■	○
10		■	■	■			■	■	■	○	■
9					■	■	■	■	○	■	■
8		■	■	■		■		○	■	■	
7		■	■	■	■		○		■	■	
6		■	■	■		○		■	■		■
5		■	■	■	○		■		■		■
4				○	■	■	■	■		■	■
3			○		■	■	■	■		■	■
2		○			■	■	■	■		■	■
1	○										
	1	2	3	4	5	6	7	8	9	10	11

Figure 5.11 Concurrency matrix from reachability tree in Figure 5.10

(4,7), and (5,7) then remain to be checked.

The selector sets to be intersected are calculated for blocks 2, 4, 5, and 7 in *quad* (refer to Figure 4.20). Since the only possible common selectors must have the */shared* prefix, they are the only members shown from the interference sets. The important members of these sets, then, are:

$S_2^{w} = \{ /shared/quit \}$
$S_4^{w} = \{ /shared/cbuff/a, /shared/cbuff/b, /shared/cbuff/c \}$
$S_5^{w} = \{ /shared/rbuff/r1, /shared/rbuff/r2 \}$
$S_7^{w} = \{ /shared/rbuff/r1, /shared/rbuff/r2 \}$

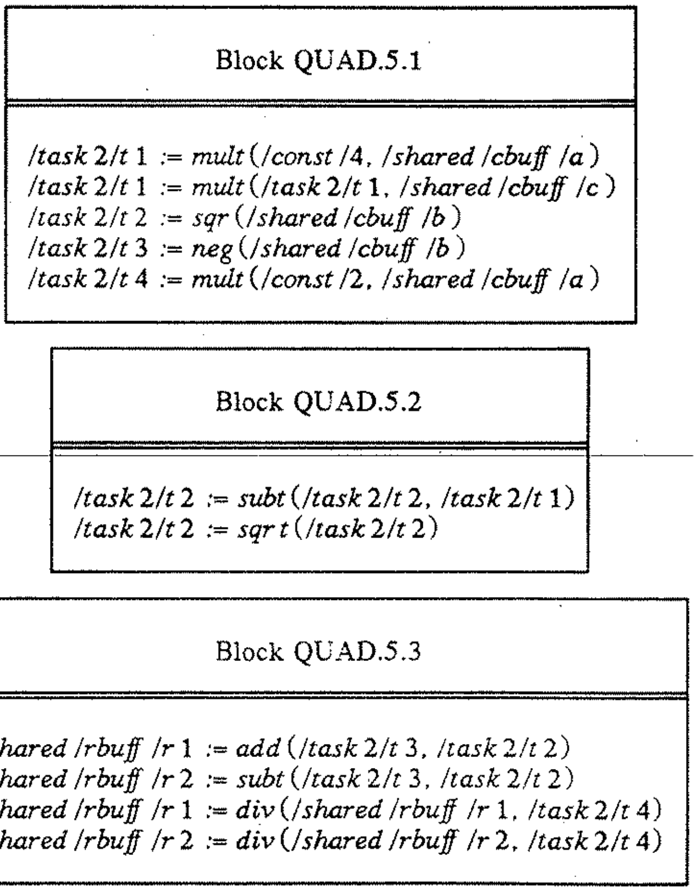
$S_2^{rr} = \varnothing$
$S_4^{rr} = \varnothing$
$S_5^{rr} = \{ /shared/cbuff/a, /shared/cbuff/b, /shared/rbuff \} \cup \{ /shared, /shared/cbuff \}$
$S_7^{rr} = \varnothing$

$S_2^{wr} = \varnothing$
$S_4^{wr} = \varnothing$
$S_5^{wr} = \{ /shared/rbuff/r1, /shared/rbuff/r2 \} \cup \{ /shared, /shared/rbuff \}$
$S_7^{wr} = \varnothing$

Given these, the only non-null interference intersections are:

$S_4^{w} \cap S_5^{rr} = \{ /shared/cbuff/a, /shared/cbuff/b \}$
$S_7^{w} \cap S_5^{rr} = \{ /shared/rbuff/r1, /shared/rbuff/r2 \}$
$S_7^{w} \cap S_5^{wr} = \{ /shared/rbuff/r1, /shared/rbuff/r2 \}$

Critical regions in blocks 4, 5, and 7 are identified based on the selectors in the interference intersections. Blocks 4 and 7 each contain a single procedure call, so the entire block in each case is the critical region. Block 5 is more complicated. The selectors in *rr* conflict between blocks 4 and 5 are */shared/cbuff/a* and */shared/cbuff/b*. Two *rr* critical regions are therefore defined in block 5 for the basic block pair (4,5). The critical region on */shared/cbuff/a* goes from the first assignment to the fifth. The critical region on */shared/cbuff/b* encompasses the third and fourth assignments. Since these regions overlap, they are coalesced into a single region. Note that the critical region associated with */shared/cbuff/b* is completely contained in the one associated with */shared/cbuff/a*.



Block QUAD.5.1

```
/task2/t1 := mult(/const/4, /shared/cbuff/a)
/task2/t1 := mult(/task2/t1, /shared/cbuff/c)
/task2/t2 := sqr(/shared/cbuff/b)
/task2/t3 := neg(/shared/cbuff/b)
/task2/t4 := mult(/const/2, /shared/cbuff/a)
```

Block QUAD.5.2

```
/task2/t2 := subt(/task2/t2, /task2/t1)
/task2/t2 := sqrt(/task2/t2)
```

Block QUAD.5.3

```
/shared/rbuff/r1 := add(/task2/t3, /task2/t2)
/shared/rbuff/r2 := subt(/task2/t3, /task2/t2)
/shared/rbuff/r1 := div(/shared/rbuff/r1, /task2/t4)
/shared/rbuff/r2 := div(/shared/rbuff/r2, /task2/t4)
```

Figure 5.12 New basic blocks formed from QUAD.5

so the coalesced critical region is exactly the greater one.

The selectors in *rr* conflict between blocks 5 and 7 are */shared/rbuff/r*1 and */shared/rbuff/r*2. Two *rr* critical regions are therefore defined in block 5 that are to be excluded with block 7. The critical region associated with */shared/rbuff/r*1 goes from the eighth assignment through the eleventh. The critical region on */shared/cbuff/b* encompasses the ninth through the twelfth assignments. These two regions also overlap and must be coalesced. Neither completely contains the other, so their union is larger than either individual region. The coalesced critical region encompassing assignments eight through twelve is the section that must be excluded with basic block 7.

The final intersection is a *wr* type between blocks 5 and 7, and the intersection set of selectors is the same as in the previous conflict. The critical regions defined by these selectors also happen to be the same, though this is only a coincidence of this particular example. No further subdivision of basic block 5 is therefore required.

Three new basic blocks, then, are formed from block 5 in *quad* from these critical regions. The new structure is shown in Figure 5.12. Of these new blocks, along with the remaining originals from Figure 4.20, block QUAD.4 must be mutually excluded with block QUAD.5.1, and block QUAD.7 must be excluded with block QUAD.5.3. The new Petri net structure is shown in Figure 5.13. It is formed from the net in Figure 4.22, with appropriate new places and transitions added to provide mutual exclusion between the critical regions. The updated functions in the control flow model are shown in Figure 5.14. Note that the two mutual exclusion places, p_{m1} and p_{m2}, are associated by the path selection function ρ with transition sets containing more than a single element. Synchronization is encapsulated by the fact that only one transition from such a set can fire at any state change. Also note that the Petri net representing the control flow for *quad* is no longer in the *free choice* class.

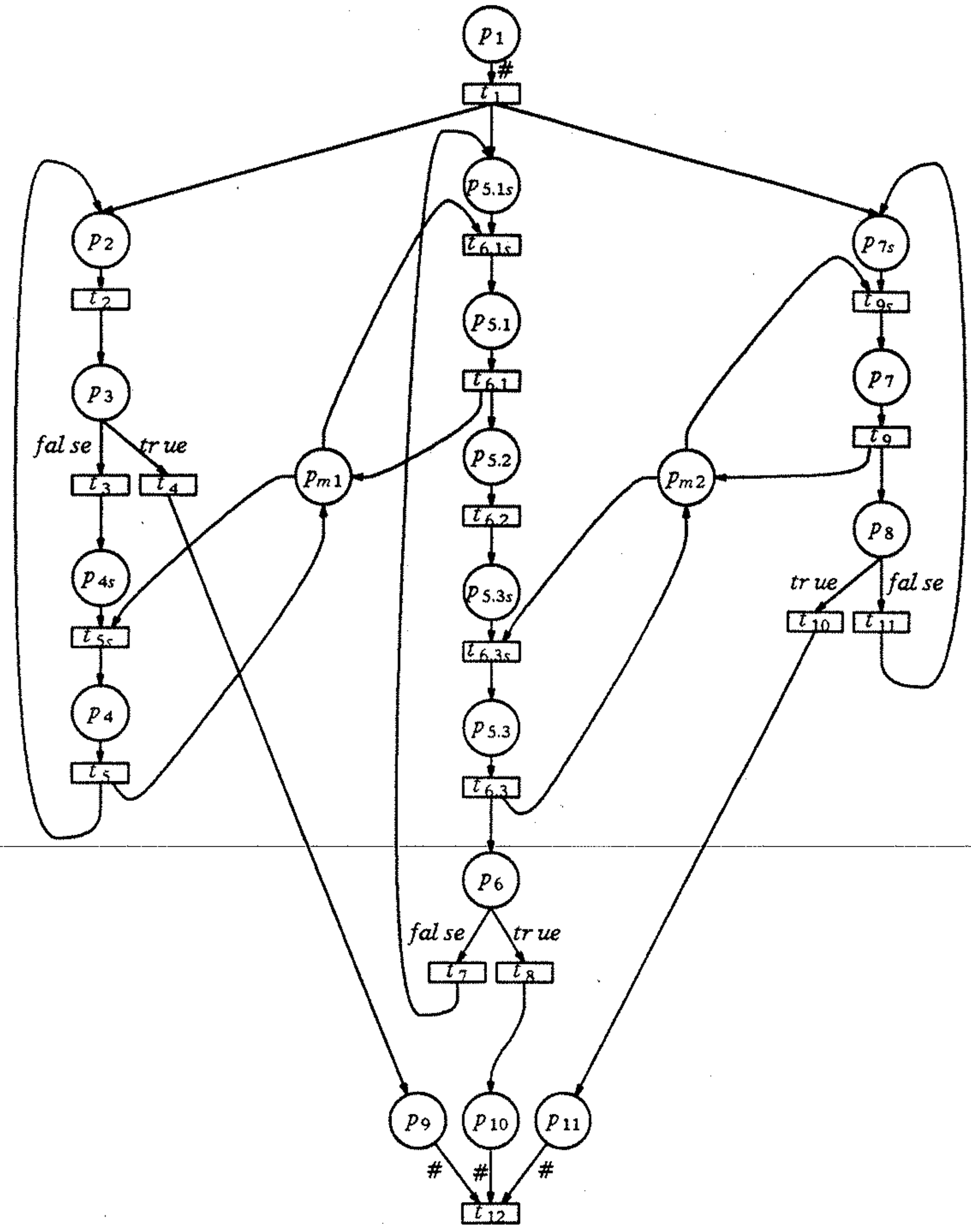


Figure 5.13 Petri net structure with mutual exclusion

place	β	σ	τ	μ_0
p_1	*Block NIL*	/	1	$\langle 1,1\rangle$
p_2	*Block QUAD.2A*	/	1	$\langle 0,0\rangle$
p_3	*Block NIL*	*/shared /quit*	1	$\langle 0,0\rangle$
p_{4s}	*Block NIL*	/	1	$\langle 0,0\rangle$
p_4	*Block QUAD.4*	/	1	$\langle 0,0\rangle$
$p_{5.1s}$	*Block NIL*	/	1	$\langle 0,0\rangle$
$p_{5.1}$	*Block QUAD.5.1*	/	1	$\langle 0,0\rangle$
$p_{5.2}$	*Block QUAD.5.2*	/	1	$\langle 0,0\rangle$
$p_{5.3s}$	*Block NIL*	/	1	$\langle 0,0\rangle$
$p_{5.3}$	*Block QUAD.5.3*	/	1	$\langle 0,0\rangle$
p_6	*Block NIL*	*/shared /quit*	1	$\langle 0,0\rangle$
p_{7s}	*Block NIL*	/	1	$\langle 0,0\rangle$
p_7	*Block QUAD.7*	/	1	$\langle 0,0\rangle$
p_8	*Block NIL*	*/shared /quit*	1	$\langle 0,0\rangle$
p_9	*Block NIL*	/	1	$\langle 0,0\rangle$
p_{10}	*Block NIL*	/	1	$\langle 0,0\rangle$
p_{11}	*Block NIL*	/	1	$\langle 0,0\rangle$
p_{m1}	*Block NIL*	/	1	$\langle 1,1\rangle$
p_{m2}	*Block NIL*	/	1	$\langle 1,1\rangle$

place	atom	ρ
p_1	#	$\{t_1\}$
p_2	#	$\{t_2\}$
p_3	false	$\{t_3\}$
p_3	true	$\{t_4\}$
p_{4s}	#	$\{t_{5s}\}$
p_4	#	$\{t_5\}$
$p_{5.1s}$	#	$\{t_{6.1s}\}$
$p_{5.1}$	#	$\{t_{6.1}\}$
$p_{5.2}$	#	$\{t_{6.2}\}$
$p_{5.3s}$	#	$\{t_{6.3s}\}$
$p_{5.3}$	#	$\{t_{6.3}\}$
p_6	false	$\{t_7\}$
p_6	true	$\{t_8\}$
p_{7s}	#	$\{t_{9s}\}$
p_7	#	$\{t_9\}$
p_8	true	$\{t_{10}\}$
p_8	false	$\{t_{11}\}$
p_9	#	$\{t_{12}\}$
p_{10}	#	$\{t_{12}\}$
p_{11}	#	$\{t_{12}\}$
p_{m1}	#	$\{t_{5s}, t_{6.1s}\}$
p_{m2}	#	$\{t_{6.3s}, t_{9s}\}$

Figure 5.14 Updated functions in the control flow model

5.5. Summary

Given the HG model of a software system described in the previous chapter, we present a method of identifying pairs of basic blocks that have the potential to execute concurrently. The solution involves calculation of a different form of reachability tree for the Petri net in the control flow model, one in which concurrent firing of transitions creates a single state change rather than the more commonly used sequence of state changes. The pairs of blocks that could execute concurrently (if the data state sequence of a computation were to select appropriate paths during execution) are identified from the states of this reachability tree. We then outline an algorithm which will determine, for a given concurrently executable pair of blocks, if they use shared data in a way that will compromise the integrity of the data state produced by their computation. If conflicts are found, critical regions are located in the blocks and alterations to the control flow model are made to force the critical regions to execute in mutual exclusion. The analysis is intended to be performed on untimed models, *i.e.*, models in which $\forall p \in P[\tau(p)=1]$. The following chapter discusses the use of timing in analysis of a software system model.

Chapter 6

TIMING OF AN HG SOFTWARE SYSTEM MODEL

In Chapter 5, the analyses applied to the HG model of a software system assumed that the times on the places were all unity. We now consider the more general problem of analysis in the case of non-unity timings. In this chapter we first present a general overview of the information that is represented in the Petri net timing for each type of PFG node. An extension to the concurrent reachability tree is then described which allows concurrent conflict detection and correction to be performed on systems with basic block times greater than one. The final sections discuss a method for generating a consistent set of timing figures for the procedures in an HG software system model. Using the extended concurrent reachability tree, we describe a technique for determining minimum and maximum execution durations for a procedure, and we give an example of how these can be used to verify the compliance of a system with time constraints.

6.1. Constructing the timing function τ

The designer of a software system (and hence its model) must ascribe durations to the various activities in the system. For some activities this is a straight-forward endeavor, but for others a bit of calculation must be performed. Some computation activities are not easily timed at all; rather, the best one can do is to provide maximum or minimum times that such an activity is expected to require. Such computations might involve an access to the external environment of an executing process, such as *receive message*, *await signal*, or *get input*. Much of our analysis, then, is for software with instructions that can be ascribed a fixed duration, but we must also consider instructions for which no exact timing is available.

The elements of the timing function in a timed Petri net are obtained according to the interpretation we apply to the various types of PFG node. Each place in the Petri net is ascribed a duration based on which PFG structure gives rise to the net component containing it, and the which activities in the modeled software are being represented. The following sections discuss system activities associated with each type of PFG node:

- **branch nodes**

  The duration of a branch node is the time required to reference the selector that is the value of the node, plus the time required to spawn any concurrent activity to follow. The exact figures naturally depend on the hardware (and software) that is to execute the system. The value arrived at in this way is ascribed to the place in the control flow model Petri net that corresponds to the particular branch node being timed.

- **procedure call nodes**

  The duration of a procedure call node is calculated to be the time that the procedure may take to execute. This value, for example, might be obtained from the PFG modeling the procedure by searching the graph for the longest path, and summing the times associated with the nodes along the path. Alternately, the time for a procedure call can simply be assigned rather than calculated. When a procedure call represents some atomic action, such as a machine instruction, the procedure is simply represented as a function with no internal structure. The time, then, is assigned according to some other information (*e.g.*, from the specifications in a hardware manual). A place corresponding to a basic block formed from a sequence of procedure call nodes is given a time in the Petri net that is the sum of the durations for the individual procedure calls.

- **join nodes**

  A join node behaves differently from the two other types of PFG node. The duration of a place in a join Petri net structure is set to the time required to execute code that determines if control has reached all incoming arcs at the join, plus the time required to execute the code needed to remove a concurrent process from activity. Some place in a join will likely receive a token before the others do. The token will age for the duration τ ascribes to that place, but then will continue to age beyond 0 into negative values until all places in the join have fully aged tokens. The greatest negative token age in a join place will then indicate the length of the *delay* which resulted from the synchronization of concurrent control paths.

6.2. The concurrent reachability tree for timed Petri nets

As we mentioned in the Chapter 3 discussion of the control flow model, the execution rule for timed Petri nets is a generalization of the rule for normal Petri nets. At each state, a set (possibly null) of enabled transitions is identified and some subset of those is chosen to fire. The simultaneous firings determine the next state. In the special case restricting the cardinality of the subset to one transition, we have the execution rule of normal Petri nets. If we further restrict all place durations in the model to one time unit, the notion of a token aging before it enables any transition is nullified and the model becomes identical to normal Petri nets.

As in the special case of normal Petri nets, the reachability tree for timed nets graphically represents the reachability set. But since the state of a timed net contains more information than the state of a normal net, the reachability tree becomes more complex. Each arc in a reachability tree is labeled with the *subset* of transitions that fire in the state change represented by the two nodes connected by the arc. A state change in a timed net may not entail any transition firings, however; if none are enabled, then the tokens in the places simply age one time unit. This passage of time is recorded in the new state, but no transitions have fired. Thus, some arcs in the reachability tree may be labeled with the *null set*, $\emptyset$. Under the concurrent form of firing rule, the tree will have leaving each node either a single null labeled arc, or as many arcs as there are maximal subsets of T_e, the enabled transitions. Maximal in this context means that, according to the semantics of net execution, as many of the enabled transitions as possible are chosen to fire. Not all enabled transitions may necessarily fire in one state change, due to conflicts over shared input places, and possible exclusion by the data state.

Since the structure and execution of normal Petri nets are special cases of the structure and execution of the timed Petri nets, we would expect the reachability trees for timed Petri nets to be a generalization of the trees for the normal nets. To see that this is indeed the case, consider the restrictions stated above that make timed nets equivalent to normal nets. If the cardinality of the set of enabled transitions chosen to fire is limited to unity, then each arc leaving a node in the tree is labeled with a set containing the single transition that produces a following state. At each node there are as many arcs leaving as there are subsets of single elements, as is the case in reachability trees for normal nets. If in addition the duration of each place is limited to one time unit, then no token is forced to sit at a place without enabling the transitions that may follow. There will then be no states at which a state change is possible but no transitions fire; hence there are no arcs in the tree labeled with the null set. So under these two restrictions, every arc in the reachability tree is labeled with one and only one transition, indicating that that transition was fired to create the represented state change.

Choosing one transition to fire (as in the normal net execution rule) seems appropriate for modeling a system when time is no factor. When execution time is to be modeled as well, however, choosing one transition is only adequate for a uniprocessor system; true concurrency in real-time must be represented by allowing simultaneous transition firings in one state change as in our timed Petri net model. The size of the subset chosen from the enabled transitions is thus a parameter of the model. One reasonable interpretation for this parameter is to let the maximum size of the subset chosen correspond to the number of physical processing units available in the host machine when the state change is taking place. For example, choosing to fire one transition per state change models a uniprocessor system. We have selected the opposite case for the analysis in this report by considering only the execution rule that chooses as many as possible of the enabled transitions to fire. This can be viewed as assuming that the number of available processing units always exceeds the number of basic blocks that need to execute, such as in the *paracomputer* described by Schwartz [60]. Since all timed Petri nets have a finite number of nodes, and since a software system model is composed of a finite number of procedure models, we are not assuming an infinite number of processors--simply more than a program can possibly need at any point in execution. The final chapter on further research discusses the ramifications of an execution model in between these two extremes, where the assumption is that some blocks may be executable but no hardware is available to execute them.

With the fire-them-all execution rule, no delays are introduced into a computation sequence that are not prescribed by the inherent synchronization of the algorithm. When a reachability tree is generated from a timed Petri net under this execution rule, its structure has a unique feature: there are no branches in the tree (*i.e.*, nodes with two or more out-arcs) unless they are a result either of a data-based decision in the algorithm (*e.g.*, at a branch node), or of a synchronization decision (*e.g.*, at a mutual exclusion place added by the conflict resolution procedure). No branches are introduced as in normal net trees from having to arbitrarily choose some transitions to fire and some to delay. The complexity of the tree is reduced from the potentially exponential growth suggested by choosing *all* subsets of a set as arcs at each node. Instead, often only *one* arc leaves a node; when more leave, the branching represents true decision making in the algorithm and not an implementation-dependent choice of which procedures get processor time and which others must wait.

6.2.1. Constructing the timed reachability tree

A node in a timed concurrent reachability tree is, as before, a marking of the Petri net. A timed marking is an extension of an untimed one: instead of an n-tuple of non-negative integers for a net with n places, we use an n-tuple of *pairs*, each pair being a non-negative integer μ_i^ν (the *number* of tokens at a place) and another (possibly negative) integer μ_i^τ (the *age* of the least recently arriving token). The timed, concurrent reachability tree is generated by much the same algorithm as for the untimed tree in Chapter 5. Some modifications are necessary, however, to properly insert ω symbols into the timed states:

- an ω symbol can only be inserted into a state at place p_i if $\mu_i^\tau = \tau(p_i)$, which is to say a token in the place is just beginning to age;
- ω symbols are only used for the μ_i^ν components of each pair, not for the age components μ_i^τ;
- When a duplicate state μ' is being sought for a state μ, the age components μ_i^τ of the places in each state are compared along with the number of tokens, *unless* the age is $\leqslant 0$, in which case it is considered to match *any* non-positive age; a non-positive token age is a sign of waiting, and should be treated as a reproducible condition.
- If a state is encountered in which no transitions are enabled, but tokens are still in the net, all with zero or negative ages, then such a state is termed a *deadlock* and is handled in the tree as a terminal state is, having no successor nodes.

The possible next states are generated from each current state in the tree by applying the concurrent execution rule for the control flow model, still with the restriction that the data state is not consulted to determine eligible transitions to fire. States are generated from the Petri net structure alone, and the age component of each place marking is handled as described in the control flow model. As states change, the token age of each place is decremented by unity until it reaches one. At this point it can either contribute to enabling a transition or it can continue to age below zero, indicating synchronization delay.

The conditions for inserting an ω symbol into μ_i^ν for a place p_i then becomes

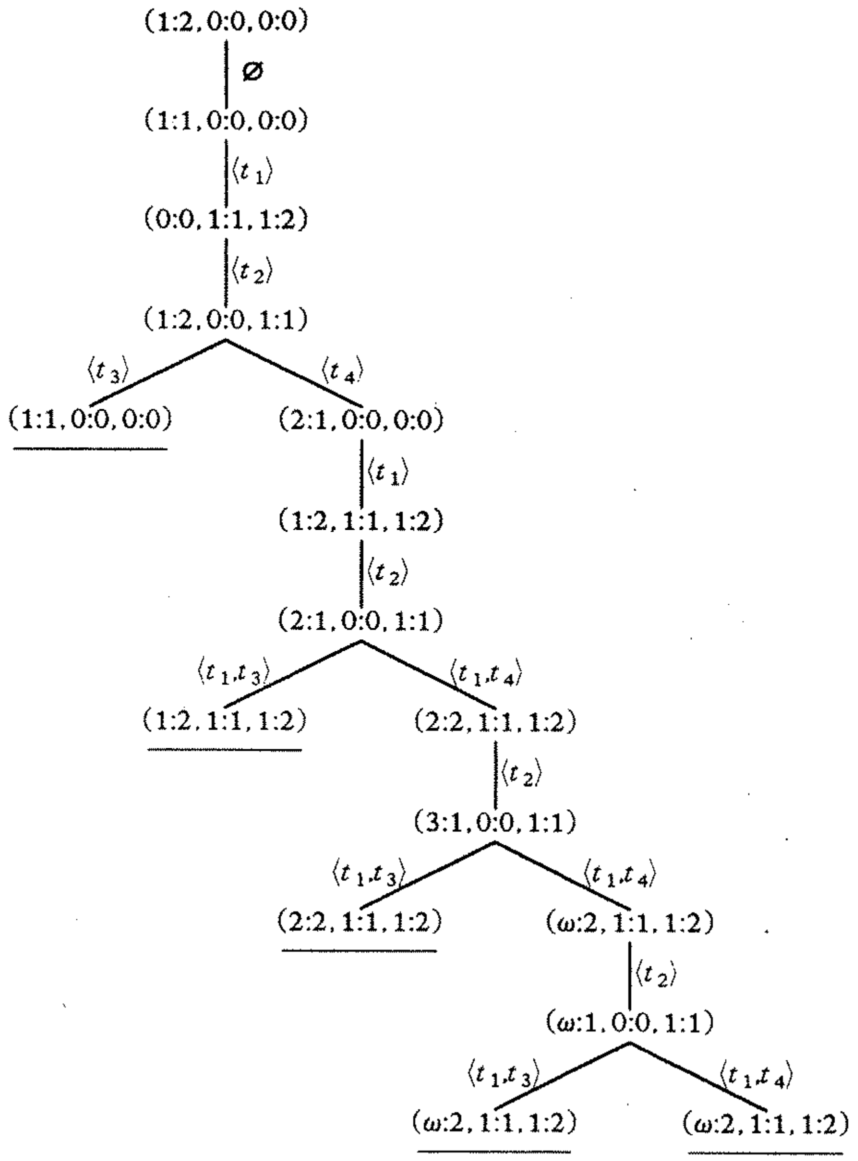


Figure 6.1 Timed concurrent reachability tree for net in Fig. 5.1

$$\exists\mu' \overset{*}{\Longrightarrow} \mu : \left| \forall p_k \left|\mu(p_k)^\nu \geqslant \mu'(p_k)^\nu\right| \wedge \ \mu(p_i)^\nu > \mu'(p_i)^\nu \wedge \mu'(p_i)^\nu > 1 \wedge \mu(p_i)^\tau = \tau(p_i) \right|$$

where $\overset{*}{\Longrightarrow}$ means "precedes in some state sequence from the initial state." Note that though the time components of a state are consulted to determine whether ω symbols should replace the token numbers, no ω symbols are ever inserted into the time components. Figure 6.1 shows the timed concurrent reachability tree for the simple Petri net structure from Figure 5.1, with the timing function $\tau=\langle 2,1,2\rangle$.

As with the algorithm for the untimed tree, the algorithm to produce the timed reachability tree can be shown to terminate:

Theorem 6-1:
The timed concurrent reachability tree is finite.

Proof: Proof by contradiction.

If the reachability tree were infinite, then since there are a bounded number of arcs leaving each node there would have to be some infinite path in the tree. Assume the existence of such a path. Since it contains no duplicate nodes, each state must be unique. A state has a finite number n of component pairs (token number and age), so an infinite number of unique states is only possible if either the token numbers grow without bound, or if the ages decrease without bound. Whenever all token ages fall below 0 the algorithm terminates a path with a deadlock, so the second alternative cannot create the infinite path.

Consider, then, all states in which the token numbers $\mu_i^\gamma \leqslant 2$. There are only a finite number of such states in which the ages are also greater than 0. Eventually the infinite path must contain a state in which some token number is 3 or greater, and it covers an earlier state. At this point the algorithm inserts an ω for the token number. The argument is now applied to states of size $n-1$. Eventually a state will be encountered in which all token numbers are ω. By definition of the algebra of ω, some following state will be a duplicate of the all-ω state, and the path will terminate. But this contradicts the assumption that the path is infinite, so the timed concurrent reachability tree must be finite.

□

Given the timed concurrent reachability tree for a Petri net, we can calculate certain execution times for the net by examining paths in the tree. The following lemma establishes the necessity and sufficiency of terminal leaves in the reachability tree for determining finite execution sequences in the Petri net.

Lemma 6-1:

Given a marked timed Petri net TP and its timed concurrent reachability tree R, there exists a path from the root of R to a terminal leaf if there exists a corresponding finite execution state sequence for TP from the initial state to a terminal state.

Proof: By construction, the paths of the reachability tree represent all possible state sequences obtained in the Petri net. A terminal state μ of the control flow model is stated in Definition 3-19 to be a marking in which no places are active, that is, in which no tokens are in the net. This implies that no transitions are enabled to fire in μ. No arcs can leave such a state in the reachability tree, making all such markings μ terminal leaves of the reachability tree. Thus a net execution sequence ending in a terminal state will produce a terminal leaf in the reachability tree.

□

Using this result, the following argument relates the length of a path in the reachability tree to the execution time required by the computation represented by the path:

Theorem 6-2:

Given a marked, timed Petri net TP, and given the reachability tree R of TP,

$$L_{\min} \leqslant E_{\min}$$

where $L_{\min}$ is the length of the shortest path from the root to a terminal node in R and $E_{\min}$ is the shortest possible execution time of TP.

Proof: Each arc in R represents a state change in the execution of TP. In the execution rule for marked timed Petri nets (in Definition 3-19), a state change occurs for the passing of a time unit; the next state is derived from the current one by first decrementing the age of each active token by one. Thus, each arc in R--each path of length one--represents an execution sequence with a duration of one time unit. By induction on the path length, a path of length of n represents an execution sequence of duration n time units. From this, with Lemma 6-1, the shortest path from the root of R to a terminal leaf represents a possible execution sequence of minimum duration. If there is an initial data state that allows this execution sequence to be realized, then $L_{\min} = E_{\min}$. Otherwise, the minimum execution time is at least as long as $L_{\min}$.

□

This result obviously applies only to trees which have at least one terminal node. If no terminal nodes exist, then a finite minimum execution time may not be determinable from the paths in the tree.

When seeking the minimum execution time of a procedure, it is sufficient to search only paths in the tree from the root to a terminal node, ignoring all duplicate nodes and ω nodes. Since the tree is generated breadth-first, no duplicate node can be on the shortest path. This is seen from the fact that if a marking μ' is found in the tree, for which the current marking μ is a duplicate, then μ must be at least as far from the root as μ' (which is generated earlier).

Further, no terminal markings can be reached from a marking containing an ω symbol. The ω represents a potentially infinite sequence of states, able to generate any number of tokens in a place. Thus, transitions will be enabled from any ω state, and hence none will be terminal. In addition, the algebra of ω dictates that $\omega \geqslant \omega$; once an ω is in a marking, it cannot be removed. Therefore, no marking leading from an ω state will be terminal.

A complementary result holds for reachability trees in which all leaves are terminal (*i.e.*, no nodes are duplicates or ω states). For a computation represented by such a tree,

$$L_{\max} \geqslant E_{\max}$$

where $L_{\max}$ is the length of the longest path in the tree and $E_{\max}$ is the longest possible execution time of the procedure. As with the minimum-time case, the inequality results from the fact that the longest path in the tree may represent a computation that is unrealizable under system execution, due to data-dependent decisions. In the presence of one or more non-terminal leaves, the maximum execution time may be infinite.

With the timed concurrent reachability tree, the same conflict detection and correction analysis as presented for untimed nets can be performed for timed nets. Some modification to the concurrency identification conditions is required: two blocks b_i and b_j

are considered potentially concurrent if in any marking μ their associated places p_i and p_j contain at least one token each, *and* the age of both current tokens is greater than 0. Formally stated, these conditions are

$$\left(\mu_i^{\nu}>0 \wedge \mu_j^{\nu}>0\right) \wedge \left(\mu_i^{\tau}>0 \wedge \mu_j^{\tau}>0\right).$$

Notice that the condition on the age components of a marking treats a block that is waiting as not executing. The timing, then, provides an added analytic advantage: conflict will only be detected in the timed nets if two blocks lie on potentially parallel paths, *and* if the execution durations cause the blocks to overlap in time. The analysis on untimed nets could make use only of potentially parallel paths in deciding whether two blocks were potentially concurrent.

6.3. Using dual timed models for constraint analysis

We turn now to the task of employing the timing information in the system model to solve problems related to real-time execution restrictions. We have assumed up to now that the net timing function τ was available for each modeled procedure in an HG system. These timings must be constructed so that the figures given as basic block durations in one procedure model make sense in terms of the computations represented by the procedures called in that basic block. Even though we are to analyze a procedure independently of the others, the reliability of the analysis naturally depends on how well the figures given in the timing correspond to the actual behavior of the rest of the system.

There are several ways to ascribe durations to procedure calls. Timing figures for each procedure call could simply be produced with no reference to the internal structure of the procedures being timed. The choice could be arbitrary, selecting different values for blocks to test the response characteristics for the structure of an algorithm under different timings. This approach is experimental--useful if a system is under development and is open to alterations. Time-critical code sections could be identified and system behavior experimented with until acceptable limits are discovered. The code in the basic blocks could then be written to fit the limits. Alternatively, timing figures can be chosen based on desired behavior—say, chosen from a distribution describing the projected average activity of the target system. The figures are still deterministic, however. Each basic block $b = \beta(p_i)$ always requires exactly τ_i units of time to execute.

As another alternative, an existing system can be modeled and measured to determine if established time limits are realizable. Timing figures under this scenario cannot be arbitrarily chosen. They must be selected based on the behavior of the existing structure. The static program and control flow models for each procedure can be consulted and the timing figure for a procedure derived by a summation of the timings for the lower level calls in the basic blocks. We wish the timings ascribed to the various procedures to be *consistent* in the following sense:

Definition 6-1: Timing of a set of named procedures
Let F be a set of named procedures. A *timing for* F is a function $\tau_F : F \rightarrow \{1,2, \cdots\}$. If $f \in F$, then $\tau_F(f)$ represents the number of time units f takes to execute.

Definition 6-2: Consistent procedure timing
Given a procedure model $\pi = \langle f, D, SP, CF \rangle$, in which $SP = \langle S, F, B \rangle$, $CF = \langle TP, \beta, \sigma, \rho \rangle$, $TP = \langle P, T, I, O, \mu_0, \tau \rangle$, and given a timing τ_F for F, then π is said to have a *timing consistent with* τ_F if
$\forall p \in P$: if $\beta(p) = b \in B$, where $b = \langle w_1, w_2, \cdots, w_n \rangle$ and $w_i = \langle f_i, A_i, R_i \rangle$, then $\tau(p) = \sum_{i=1}^{n} \tau_F(f_i)$.

Definition 6-3: Consistent system timing
Given an HG software system model $SS = \langle \hat{F}, \Pi, \pi_0 \rangle$, and given a timing τ_F for the set of procedures $F = \hat{F} \cup \Pi$, then SS has a *timing consistent with* τ_F if each procedure model $\pi \in \Pi$ has a timing consistent with τ_F.

These definitions enable a modeler to decide when the individual system components describe an integrated system that makes sense in a model of real-time. A procedure π with a consistent timing has duration figures ascribed to each basic block that agree with the durations calculated from the other models describing the procedures called in the block. An inconsistent timing results if the durations of the procedure calls in a basic block are calculated and summed from the other control flow models in the system, giving a figure in conflict with the duration ascribed that particular block in the control flow model for procedure π. Such an inconsistency says that the higher-level, more abstract view of the procedure (given by π) does not present the same computation as the cumulative view from one level down (as given by the calculations from the other component models).

Real software procedures rarely exhibit deterministic behavior in the durations of their successive invocations. Data dependent decisions can shorten some executions and lengthen others. The HG model of a software system, however, uses deterministic timings; a basic block in a procedure model cannot be said to exhibit a range of durations. One way to make sense of this restriction in analyzing software is to calculate two, dual models from one software system: one model is a minimum-timed model, in which the duration of each Petri net place represents the fastest possible execution of the associated basic block; the dual model is a maximum-timed one, in which all place times represent the slowest execution of their associated basic blocks. Any actual execution time of the software system must fall in the range created by the upper and lower bounds offered by the two models. The following sections discuss the ramifications of using these dual timings for a complete software system model.

6.4. Timing basic components

As mentioned in the first section of this chapter, the basic components of a PFG can each be ascribed a duration based on the component's type and the composition of the component's associated basic block. The duration of a basic block node sequence is found by $\sum n_\pi * \tau_F(\pi)$, over all procedures π called in the block, with each π being called n_π times. Note that the *procedures* are timed, in that any one call to a procedure π is indistinguishable from another. The duration of a branch node is the time required to access the data state at the node selected, and the time needed to create concurrent execution paths, if any. The duration of each place in the net representation of a join node is the time required to remove concurrent control paths. The basic blocks associated with these nodes represent overhead computations; the durations are determined as for other basic blocks.

6.5. Timing delays

In the context of a Petri net, waiting is defined to be the delay of a token from progressing through the net. Waiting can occur for two reasons. A token may be delayed because it arrives at a place which already contains another token. Under the execution rule employed in the HG model, only one token at a time may age at a place because a place represents a resource. When a second token arrives, the processing capacity modeled by the place is occupied by the earlier token. This form of waiting is therefore termed *resource delay*. The *pipeline wait* presents a form of resource delay and is discussed in more detail below.

A token may also be delayed at a place *after* the associated basic block has executed. This occurs when some, but not all, input places to a transition are marked, and the firing of the transition is thereby held up. This form of waiting is termed *synchronization delay* because the control paths containing these input places are synchronized at the transition. *Join waiting* and *mutual exclusion waiting*, two ways that synchronization delay can occur in the HG model, are discussed in more detail below.

At any place p in a Petri net which is part of an HG system model, the following formulae hold for the minimum and maximum times that a token may reside at p:

$$T^p_{\min} = E^p_{\min} + R^p_{\min} + S^p_{\min} \tag{1}$$

$$T^p_{\max} = E^p_{\max} + R^p_{\max} + S^p_{\max} \tag{2}$$

In these equations, T is the total residence duration of the token, E is the time required for the execution of the basic block associated with p, R is the resource delay, and S is the synchronization delay. The figures for E are obtained from the timing functions τ for the dual HG models (minimum-timed and maximum-timed) of a system. The delay figures are obtained as explained in the following discussion. In some situations, delay time may be zero and not contribute to a token's residence time; in other situations, a delay may be unbounded, denoted by ∞.

6.5.1. Resource delay

Resource delay cannot occur at any place p_i in a net in which all markings in the reachability tree have $\mu_i^\nu \leqslant 1$. Conversely, if there exists a tree node such that $\mu_i^\nu > 1$, then resource delay might occur, depending on whether or not the data will allow the path leading to that state to be executed. At a place p that can have resource delay, the minimum waiting time a token can incur is given by

$$R_{\min}^p = \left(E_{\min}^p + S_{\min}^p\right) * (n_{\min}^p - 1)$$

where $n_{\min}^p$ is the *smallest* positive number of tokens that p may contain (obtained from the states in the reachability tree). The maximum resource delay time for a token at p is given by a similar equation,

$$R_{\max}^p = \left(E_{\max}^p + S_{\max}^p\right) * (n_{\max}^p - 1)$$

but with $n_{\max}^p$ here being the *largest* number of tokens found for p in the reachability tree nodes. If $n_{\max}^p = \omega$, then $R_{\max}^p = \infty$.

6.5.2. Pipeline waiting

Pipeline waiting is an extension of resource delay to a series of place/transition pairs, termed a *pipeline* (see Figure 4.14). Each place in a pipeline, other than the head place p', can have only a single input arc; transitions likewise have single input arcs. Tokens then enter a pipeline at p', and filter down through it, eventually leaving via the firing of the last transition in the pipe. Since no transition in a pipeline has more than one input place, there can be no synchronization delay and the S terms are zero in the token residence time equations (1) and (2). The maximum total resource delay that a token will incur, then, in a pipeline is given by

$$R_{\max} = \sum_{p \in pipe} E^p_{\max} * (n^p_{\max} - 1)$$

where $n^p_{\max}$ is the maximum number of tokens that can be in place p. The total block execution time for a token in the pipe is given by

$$E_{\max} = \sum_{p \in pipe} E^p_{\max}.$$

To get any analytical formulae for the total residence time extrema a token may have in a pipeline, we define a simplified pipeline structure:

Definition 6-4: Normal pipeline
A *normal pipeline* is a pipeline Petri net structure in which
$\tau(p') \geqslant \tau(p)$
for the initial place p' and any other place p in the pipeline.

In a normal pipeline, the execution time of the basic block associated with the first place is at least as long as that of any other block represented in the pipe[1]. All resource waiting therefore occurs in a normal pipeline at the first place p'. Since places after p' in the pipe have delays that are no longer than the first, no tokens accumulate at these places and no resource delay is created at them. As mentioned earlier, pipelines also have no synchronization delay. Equation (1) for the minimum total token residence time reduces to

$$T^{pipe}_{\min} = R^{p'}_{\min} + \sum_{p \in pipe} E^p_{\min} = E^{p'}_{\min} * (n^{p'}_{\min} - 1) + \sum_{p \in pipe} E^p_{\min}$$

where $n^{p'}_{\min}$ is the minimum number of tokens that can concurrently reside at place p', obtained from the timed reachability tree. In a similar manner, the maximum total time for a token to pass through a pipeline is obtained from the maximum resource delay and the sum of the execution times. This is given by equation (2), which reduces to

[1]Though this may seem restrictive, note that any arbitrary pipeline can be treated as a sequence of normal pipelines.

$$T^{pipe}_{\max} = R^{p'}_{\max} + \sum_{p \in pipe} E^{p}_{\max} = E^{p'}_{\max} * (n^{p'}_{\max} - 1) + \sum_{p \in pipe} E^{p}_{\max}$$

where $n^{p'}_{\max}$ is the maximum number of tokens that can concurrently reside at place p', also obtained from the timed reachability tree.

6.5.3. Join waiting

Join waiting is a form of synchronization delay, though some resource delay can be involved as well (as it can at *any* place). A token arriving at a place that is part of a join component must wait until all the join places are marked and fully aged before a control path may continue on. The length of this wait is variable. We wish to know $T^{p}_{\max}$ and $T^{p}_{\min}$, the extrema in residence times for a token at a join place p. Formulae (1) and (2) give these figures, though we are left with determining the right hand terms. The resource delay time at p is the same as for any other place:

$$R^{p}_{\min} = \left[E^{p}_{\min} + S^{p}_{\min}\right] * (n^{p}_{\min} - 1)$$

$$R^{p}_{\max} = \left[E^{p}_{\max} + S^{p}_{\max}\right] * (n^{p}_{\max} - 1)$$

where, as before, the respective execution times and token numbers come from the dual HG system models and their timed concurrent reachability trees. Combining these with equations (1) and (2), we see that the total token residence times are given by

$$T^{p}_{\min} = \left[E^{p}_{\min} + S^{p}_{\min}\right] * n^{p}_{\min}$$

$$T^{p}_{\max} = \left[E^{p}_{\max} + S^{p}_{\max}\right] * n^{p}_{\max}$$

where now only the synchronization delays $S^{p}_{\max}$ and $S^{p}_{\min}$ remain to be determined.

These delays are more difficult to obtain analytically, as they require knowledge of events on control paths other than the one containing place p. We therefore employ the reachability tree to obtain the bounds on synchronization delay. During execution of the Petri net, the age of a delayed token is decremented past zero and becomes increasingly negative, thereby giving a measure of the number of state changes spent in a synchronization delay. To find $S_{\min}$, search the nodes of the tree for states which represent the final state in a waiting period at p. These are states in which p is marked, the token age is zero or less, and in at least one of the *following* states the age is reset to either $\tau(p)$, or to 0 with no tokens remaining, indicating that the token has been consumed by a transition firing. Let $\alpha^p_{\min}$ be the numerically greatest $\mu(p)^\tau$ from these states such that $\mu(p)^\tau \leqslant 1$. Then the minimum synchronization delay is given by

$$S^p_{\min} = |\alpha^p_{\min} - 1|.$$

The maximum synchronization delay is likewise found by isolating the same final delay states from the tree, and then choosing $\alpha^p_{\max}$ to be the numerically smallest $\mu(p)^\tau \leqslant 1$. Then, the maximum synchronization delay is given by

$$S^p_{\max} = |\alpha^p_{\max} - 1|.$$

Note that an exception to this exists. If the tree contains any ***deadlocked*** states in which p is marked, then $S^p_{\max} = \infty$. Also, if there are no wait-ending states, so that no $\alpha^p_{\max}$ can be found, then $S^p_{\max} = \infty$.

6.5.4. Mutual exclusion waiting

Mutual exclusion waiting occurs at the mutual exclusion net structures created by the procedure described in Chapter 5 for correction of conflicts on shared data. It is handled in much the same way as join waiting, with the complication of potential starvation being introduced. As indicated in the earlier discussion, without mutual exclusion structures the Petri nets in an HG system model are *free choice*. No transitions in free choice nets compete for input tokens unless they share a single place as their sole input. Mutual exclusion structures change the nature of a net, and this is reflected in the timing analysis.

Consider a mutual exclusion net structure consisting of two place/transition pairs, p/t and p'/t', and an extra marked place p_{mutex} which is an input to both t and t'. The minimum time a token may reside at place p is the same as for the join wait, that is

$$T^p_{\min} = \left(E^p_{\min} + S^p_{\min}\right) * n^p_{\min}$$

where $S^p_{\min} = |\alpha^p_{\min}-1|$ and $\alpha^p_{\min}$ is the numerically greatest $\mu(p)^\tau \leqslant 1$ in the wait-ending states of the reachability tree.

The maximum residence time for a token at p is likewise similar to join waiting, and is given by

$$T^p_{\max} = \left(E^p_{\max} + S^p_{\max}\right) * n^p_{\max}$$

where $S^p_{\max} = |\alpha^p_{\max}-1|$ and $\alpha^p_{\max}$ is the numerically least $\mu(p)^\tau \leqslant 1$ in the wait-ending states of the reachability tree. Because execution of a mutual exclusion structure possibly entails choosing one of two competing transitions to fire, net structures can be created which have execution paths that never choose one of the transitions, creating a partial deadlock by starvation. In these cases, the maximum token residence time at each place in the structure would be $T^p_{\max} = \infty$. The reachability tree can be used to detect such starvations. The algorithm for generating the timed tree treats all waiting as equivalent. Thus, if a state is generated in which a token in place p has been delayed for, say, five state changes (indicated by $\mu(p)^\tau = -4$) then for the purposes of finding duplicate states already in the tree it will match with any state having $\mu(p)^\tau \leqslant 0$. So, if one of the wait-ending states for p found in the tree is a duplicate state, then there exists a cycle from the earlier waiting state to the later one, along which the token is always delayed. The fact that it is a cycle means that the waiting at p is endless on that particular execution path.

6.6. Timing acyclic PFGs

An acyclic PFG is the analytically simplest form of computation representation. Since there are no loops in the graph, the timed concurrent reachability tree for the Petri net formed from it will have no duplicate nodes and no ω nodes. Essentially, the reachability tree will contain complete information about the computation. The computations represented by acyclic PFGs are probably not common in real systems, but they are the only form of computation for which the actual minimum and maximum times are both always finite and can both always be found.

If there are no delays in an acyclic PFG, then the minimum and maximum times are easily calculated from the shortest and longest paths in the reachability tree. In the presence of delays, the minimum and maximum execution times are still found from paths in the reachability tree, but the possibility of encountering deadlock states is introduced. The tree generation algorithm detects and marks full deadlocks. Partial deadlocks are discovered as mentioned in the previous section on synchronization delay.

6.7. Timing loops

PFG's with cycles (loops) are somewhat harder to handle satisfactorily. In the most general case, cyclic computations cannot be completely analyzed automatically in the HG model theory, though some results can be obtained from the reachability tree in conjunction with its generating Petri net. For example, consider the timed reachability tree shown in Figure 6.1. Since it has no terminal state leaves, no finite minimum execution time can be found (and hence no finite maximum either). The best that can be done with such a situation is to try and bound a portion of the computation, say one loop execution. Then, to establish any bounds on the total execution time it is necessary to have bounds *a priori* on the number of traversals of each loop in the computation. These bounds may come from language constructs, like *for loops*. The maximum execution time for such a loop is given by

$$T_{\max} = ub * T_{\max}^{body}$$

where $T_{\max}^{body}$ is the maximum execution time of the loop body and ub is the maximum number of traversals of the loop. Conversely, the minimum execution time is given by

$$T_{\min} = lb * T_{\min}^{body}$$

where $T_{\min}^{body}$ is the minimum execution time of the loop body and lb is the minimum number of traversals of the loop. The figures for $T_{\max}^{body}$ and $T_{\min}^{body}$ are obtained by the previously described procedures for determining time bounds for basic components, delays, and acyclic computations, along with this procedure for timing any interior loops.

It should be noted that timing loops requires information not necessarily found in the reachability tree. The tree for a Petri net with cycles may contain duplicate nodes and ω states. Associating paths and portions of paths in the reachability tree with cycles and sections of cycles in the generating Petri net is not easily done. Some loops can be identified in the reachability tree by looking for marking and successive remarking of places that are identified from the net as the heads of loops. The duration of such a loop's body can then be determined from the path length between the states that represent the initial loop execution and the successive execution. It appears that further research in this area would be fruitful.

Another attack to timing loops is to "unroll" the loops as far as the maximum number of traversals (or minimum) and then generate the tree from the expanded net. This approach, however, causes a potentially large increase in the number of places in the net and a correspondingly large expansion of the state space represented by the reachability tree. The thrust of the research to this point has been to keep the Petri net as small as possible to prevent explosive state space growth.

6.8. Timing recursive calls

Recursive procedure calls violate the way in which we are using the HG system modeling theory. We have assumed that consistent timings for each procedure model (in the sense of Definition 6-2) can be developed only if the timings of lower level procedures are known. This assumption gives rise to the dual model approach to developing minimum and maximum time bounds on system execution. Since recursive procedure calls take obviously differing amounts of time (less time for each successive recursion) the existing model structure cannot deal with them. It is not obvious how to develop a consistent timing for a procedure that calls itself, thereby requiring knowledge of the very duration being calculated. Recursion is therefore assumed to be absent from software that is modeled with the HG formalism. Adapting the techniques described in this report to recursion is a topic for future research.

6.9. Deadlock detection

A deadlock is a state of the Petri net in which at least one token exists but no transitions are enabled, and in which no transitions will become enabled in the future. As stated in Definition 3-19, total deadlock exists when the set of active places, P_a, is non-empty, but the timing component μ_i^T of each active place p_i is negative or zero; thus no transitions are enabled. In this situation, the system will continue to age its tokens indefinitely with no firings. This form of deadlock is easily detected as the reachability tree is constructed; when such a state is created it is left as a leaf in the tree and tagged a deadlock.

A state μ contains a *partial deadlock* if there exists a transition or set of transitions with at least one input place marked[2], but that will never be enabled in any state μ' following μ. Certain potential partial deadlocks can be identified from the reachability tree by appealing to the semantics of the PFG nodes that give rise to some of the places in a Petri net. Partial deadlock in the PFG can occur at join nodes, which give rise to Petri net components in which several places serve as input to a single transition. If control passes down one of the arcs leading to a join node, but not eventually down all the others, the computation is stalled at the synchronization point.

Such a deadlock can be caused in several ways. One is that the failure to join is a natural possibility given the structure of the computation. For instance, one of the arcs entering the join is the exit from a loop, which may be non-terminating. Another way is from an incorrectly written algorithm, one in which non-concurrent control paths are accidentally joined. For example, consider a branch node having two out-arcs labeled

[2]For this analysis we ignore mutual exclusion places, as introduced in Chapter 5. The deadlock detection is assumed to be performed on Petri nets *before* the shared data conflicts are identified and repaired.

with different atoms. Further down each path from the branch, a join node has both alternative paths as in-arcs. In terms of net markings, a partial deadlock implies that a state μ exists in the reachability tree in which tokens reside at some of the places representing the join. There exist, however, no paths from μ in the tree along which a state μ' is found in which all the join places are marked.

It should be noted that the reachability tree will indeed evidence all deadlocks that exists in a computation, but at the expense of perhaps displaying some deadlocks that may not actually be possible during system execution. The data state is consulted during system execution to direct the selection of transitions to fire, but it is not employed in generating the reachability tree. Any reachability tree may then contain some execution paths which are possible from the net structure alone, but are precluded by data-dependent decisions during system execution. If a deadlock happens to be on such a path, then it will appear from analysis of the reachability tree to be a problem when it is not.

6.10. Timing complete procedures and systems

Given an HG software system model, each procedure can be timed by applying the individual methods described in the previous sections as the appropriate structures are encountered in the Petri net for a procedure, using the timing figures previously obtained for lower level procedures. We assume that timing figures exist *a priori* for some procedures, specifically the primitive procedures. These represent perhaps hardware instructions, for which manuals give the minimum and maximum execution durations. In the assumed absence of recursion, the procedure calling structure is a tree. With timing figures for the primitive procedures, the minimum and maximum times for the procedures which call the primitives can be calculated as follows.

Consider a procedure π which calls only procedures in $\hat{F}$, the primitive procedures. Consider also a timing $\tau_{\hat{F}}$ for $\hat{F}$, defined as $\tau_{\hat{F}} : \hat{F} \rightarrow \{1, 2, \cdots\}$. Then a consistent timing $\tau_{\hat{F}}'$ for the set of procedures $\hat{F} \cup \{\pi\}$ can be constructed by first constructing the net timing τ for the Petri net representing the computation performed by π. Each place in the net is given a consistent duration as described in definition 6-2. The time for each p is

$$\tau(p) = \sum_{i=1}^{n} \tau_{\hat{F}}(f_i)$$

where $\beta(p) = \langle w_1, w_2, \cdots, w_n \rangle$, and each $w_i = \langle f_i, A_i, R_i \rangle$. Then the timed concurrent reachability tree is constructed for π, and from it a value $\tau_{\hat{F}}'(\pi)$ is determined, depending on whether the system timing being generated is a minimum or a maximum. The process is repeated for the remaining untimed procedures in the system, using $\tau_{\hat{F}}'$ as the new initial timing $\tau_{\hat{F}}$.

6.11. Extended example (continued)

We continue to consider the Petri net shown in Figure 4.22, but now with a timing function of $\tau = \langle 1,2,1,3,5,1,2,1,1,1,1 \rangle$. The generating algorithm for the timed concurrent reachability tree produces the tree of 79 states shown in Figure 6.2. The concurrency matrix constructed from this tree is given in Figure 6.3. Notice how considering activity duration has eliminated some concurrent block pairs from the untimed example of Figure 5.11 and has added others that were not previously overlapping. For example, in the untimed net blocks 3 and 11 were potentially concurrent whereas they are not in this particular timed net. Conversely, blocks 5 and 8 were not potentially concurrent in the untimed system, whereas with the addition of durations they are seen to be overlapping.

Under this net timing τ the same sharing conflicts exist as in the untimed example, since block pairs $\langle 4,5 \rangle$ and $\langle 5,7 \rangle$ are still potentially concurrent. The Petri net structure is altered as previously shown in Figure 5.13; the basic blocks in the altered model are therefore the same as in the untimed model.

A minimum execution time can be calculated for the procedure *quad*, but since duplicate and ω nodes exist in the reachability tree, no absolute maximum time can be

```
s1(1:1 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0) inter
 --1--> s2
s2(0:0 1:2 0:0 0:0 1:5 0:0 1:2 0:0 0:0 0:0 0:0) inter
 --e--> s3
s3(0:0 1:1 0:0 0:0 1:4 0:0 1:1 0:0 0:0 0:0 0:0) inter
 --2-9--> s4
s4(0:0 0:0 1:1 0:0 1:3 0:0 0:0 1:1 0:0 0:0 0:0) inter
 --4-11--> s5
 --4-10--> s34
 --3-11--> s36
 --3-10--> s78
s5(0:0 0:0 0:0 0:0 1:2 0:0 1:2 0:0 1:1 0:0 0:0) inter
 --e--> s6
s6(0:0 0:0 0:0 0:0 1:1 0:0 1:1 0:0 1:0 0:0 0:0) inter
 --6-9--> s7
s7(0:0 0:0 0:0 0:0 0:0 1:1 0:0 1:1 1:-1 0:0 0:0) inter
 --8-11--> s8
 --8-10--> s15
 --7-11--> s17
 --7-10--> s32
s8(0:0 0:0 0:0 0:0 0:0 0:0 1:2 0:0 1:-2 1:1 0:0) inter
 --e--> s9
s9(0:0 0:0 0:0 0:0 0:0 0:0 1:1 0:0 1:-3 1:0 0:0) inter
 --9--> s10
s10(0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:1 1:-4 1:-1 0:0) inter
 --11--> s11
 --10--> s13
s11(0:0 0:0 0:0 0:0 0:0 0:0 1:2 0:0 1:-5 1:-2 0:0) inter
 --e--> s12
s12(0:0 0:0 0:0 0:0 0:0 0:0 1:1 0:0 1:-6 1:-3 0:0) dup s9
s13(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:-5 1:-2 1:1) inter
 --12--> s14
s14(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0) term
s15(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:-2 1:1 1:1) inter
 --12--> s16
s16(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0) term
s17(0:0 0:0 0:0 0:0 1:5 0:0 1:2 0:0 1:-2 0:0 0:0) inter
 --e--> s18
s18(0:0 0:0 0:0 0:0 1:4 0:0 1:1 0:0 1:-3 0:0 0:0) inter
 --9--> s19
s19(0:0 0:0 0:0 0:0 1:3 0:0 0:0 1:1 1:-4 0:0 0:0) inter
 --11--> s20
 --10--> s22
s20(0:0 0:0 0:0 0:0 1:2 0:0 1:2 0:0 1:-5 0:0 0:0) inter
 --e--> s21
s21(0:0 0:0 0:0 0:0 1:1 0:0 1:1 0:0 1:-6 0:0 0:0) dup s6
s22(0:0 0:0 0:0 0:0 1:2 0:0 0:0 0:0 1:-5 0:0 1:1) inter
 --e--> s23
s23(0:0 0:0 0:0 0:0 1:1 0:0 0:0 0:0 1:-6 0:0 1:0) inter
 --6--> s24
s24(0:0 0:0 0:0 0:0 0:0 1:1 0:0 0:0 1:-7 0:0 1:-1) inter
 --8--> s25
 --7--> s27
s25(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:-8 1:1 1:-2) inter
 --12--> s26
s26(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0) term
s27(0:0 0:0 0:0 0:0 1:5 0:0 0:0 0:0 1:-8 0:0 1:-2) inter
 --e--> s28
s28(0:0 0:0 0:0 0:0 1:4 0:0 0:0 0:0 1:-9 0:0 1:-3) inter
 --e--> s29
s29(0:0 0:0 0:0 0:0 1:3 0:0 0:0 0:0 1:-10 0:0 1:-4) inter
 --e--> s30
s30(0:0 0:0 0:0 0:0 1:2 0:0 0:0 0:0 1:-11 0:0 1:-5) inter
 --e--> s31
s31(0:0 0:0 0:0 0:0 1:1 0:0 0:0 0:0 1:-12 0:0 1:-6) dup s23
s32(0:0 0:0 0:0 0:0 1:5 0:0 0:0 0:0 1:-2 0:0 1:1) inter
 --e--> s33
s33(0:0 0:0 0:0 0:0 1:4 0:0 0:0 0:0 1:-3 0:0 1:0) dup s28
s34(0:0 0:0 0:0 0:0 1:2 0:0 0:0 0:0 1:1 0:0 1:1) inter
 --e--> s35
s35(0:0 0:0 0:0 0:0 1:1 0:0 0:0 0:0 1:0 0:0 1:0) dup s23
s36(0:0 0:0 0:0 1:3 1:2 0:0 1:2 0:0 0:0 0:0 0:0) inter
 --e--> s37
s37(0:0 0:0 0:0 1:2 1:1 0:0 1:1 0:0 0:0 0:0 0:0) inter
 --6-9--> s38
s38(0:0 0:0 0:0 1:1 0:0 1:1 0:0 1:1 0:0 0:0 0:0) inter
 --5-8-11--> s39
 --5-8-10--> s63
 --5-7-11--> s65
 --5-7-10--> s66
s39(0:0 1:2 0:0 0:0 0:0 0:0 1:2 0:0 0:0 1:1 0:0) inter
 --e--> s40
s40(0:0 1:1 0:0 0:0 0:0 0:0 1:1 0:0 0:0 1:0 0:0) inter
 --2-9--> s41
s41(0:0 0:0 1:1 0:0 0:0 0:0 0:0 1:1 0:0 1:-1 0:0) inter
 --4-11--> s42
 --4-10--> s44
 --3-11--> s46
 --3-10--> s61
s42(0:0 0:0 0:0 0:0 0:0 0:0 1:2 0:0 1:1 1:-2 0:0) inter
 --e--> s43
s43(0:0 0:0 0:0 0:0 0:0 0:0 1:1 0:0 1:0 1:-3 0:0) dup s9
```

Figure 6.2: Concurrent reachability tree for example timed Petri net

```
s44(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:1 1:-2 1:1) inter
 --12--> s45
s45(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0) term
s46(0:0 0:0 0:0 1:3 0:0 0:0 1:2 0:0 0:0 1:-2 0:0) inter
 --e--> s47
s47(0:0 0:0 0:0 1:2 0:0 0:0 1:1 0:0 0:0 1:-3 0:0) inter
 --9--> s48
s48(0:0 0:0 0:0 1:1 0:0 0:0 0:0 1:1 0:0 1:-4 0:0) inter
 --5-11--> s49
 --5-10--> s51
s49(0:0 1:2 0:0 0:0 0:0 0:0 1:2 0:0 0:0 1:-5 0:0) inter
 --e--> s50
s50(0:0 1:1 0:0 0:0 0:0 0:0 1:1 0:0 0:0 1:-6 0:0) dup s40
s51(0:0 1:2 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:-5 1:1) inter
 --e--> s52
s52(0:0 1:1 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:-6 1:0) inter
 --2--> s53
s53(0:0 0:0 1:1 0:0 0:0 0:0 0:0 0:0 0:0 1:-7 1:-1) inter
 --4--> s54
 --3--> s56
s54(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:1 1:-8 1:-2) inter
 --12--> s55
s55(0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0) term
s56(0:0 0:0 0:0 1:3 0:0 0:0 0:0 0:0 0:0 1:-8 1:-2) inter
 --e--> s57
s57(0:0 0:0 0:0 1:2 0:0 0:0 0:0 0:0 0:0 1:-9 1:-3) inter
 --e--> s58
s58(0:0 0:0 0:0 1:1 0:0 0:0 0:0 0:0 0:0 1:-10 1:-4) inter
 --5--> s59
s59(0:0 1:2 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:-11 1:-5) inter
 --e--> s60
s60(0:0 1:1 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:-12 1:-6) dup s52
```

```
s61(0:0 0:0 0:0 1:3 0:0 0:0 0:0 0:0 0:0 1:-2 1:1) inter
 --e--> s62
s62(0:0 0:0 0:0 1:2 0:0 0:0 0:0 0:0 0:0 1:-3 1:0) dup s57
s63(0:0 1:2 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:1 1:1) inter
 --e--> s64
s64(0:0 1:1 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:0 1:0) dup s52
s65(0:0 1:2 0:0 0:0 1:5 0:0 1:2 0:0 0:0 0:0 0:0) dup s2
s66(0:0 1:2 0:0 0:0 1:5 0:0 0:0 0:0 0:0 0:0 1:1) inter
 --e--> s67
s67(0:0 1:1 0:0 0:0 1:4 0:0 0:0 0:0 0:0 0:0 1:0) inter
 --2--> s68
s68(0:0 0:0 1:1 0:0 1:3 0:0 0:0 0:0 0:0 0:0 1:-1) inter
 --4--> s69
 --3--> s71
s69(0:0 0:0 0:0 0:0 1:2 0:0 0:0 0:0 1:1 0:0 1:-2) inter
 --e--> s70
s70(0:0 0:0 0:0 0:0 1:1 0:0 0:0 0:0 1:0 0:0 1:-3) dup s23
s71(0:0 0:0 0:0 1:3 1:2 0:0 0:0 0:0 0:0 0:0 1:-2) inter
 --e--> s72
s72(0:0 0:0 0:0 1:2 1:1 0:0 0:0 0:0 0:0 0:0 1:-3) inter
 --6--> s73
s73(0:0 0:0 0:0 1:1 0:0 1:1 0:0 0:0 0:0 0:0 1:-4) inter
 --5-8--> s74
 --5-7--> s76
s74(0:0 1:2 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:1 1:-5) inter
 --e--> s75
s75(0:0 1:1 0:0 0:0 0:0 0:0 0:0 0:0 0:0 1:0 1:-6) dup s52
s76(0:0 1:2 0:0 0:0 1:5 0:0 0:0 0:0 0:0 0:0 1:-5) inter
 --e--> s77
s77(0:0 1:1 0:0 0:0 1:4 0:0 0:0 0:0 0:0 0:0 1:-6) dup s67
s78(0:0 0:0 0:0 1:3 1:2 0:0 0:0 0:0 0:0 0:0 1:1) inter
 --e--> s79
s79(0:0 0:0 0:0 1:2 1:1 0:0 0:0 0:0 0:0 0:0 1:0 ) dup s72
```

Numbers on an arrow indicate transitions that fire to cause a change from the state (marking) above the arrow to the state number following the arrow. Multiple arrows below a state indicate alternative sets of transitions that may fire from that state.

Figure 6.2 Concurrent reachability tree for example timed Petri net (continued)

11		■		■	■				■	■	○
10		■					■			○	■
9					■		■		○		■
8			■	■	■	■		○			
7		■		■	■		○		■	■	
6				■		○		■			
5		■	■	■	○		■	■	■		■
4				○	■	■	■	■			■
3			○		■			■			
2		○			■		■			■	■
1	○										
	1	2	3	4	5	6	7	8	9	10	11

Figure 6.3 Concurrency matrix for example timed Petri net

obtained. The terminal states in the reachability tree are $s\,14, s\,16, s\,26, s\,45$, and $s\,55$. All represent the same net configuration, that of no tokens remaining in the net. The shortest path to one of these terminal states is the state sequence

$$s\,1, s\,2, s\,3, s\,4, s\,5, s\,6, s\,7, s\,15, s\,16$$

which has length 8, representing an execution sequence requiring 8 time units. This execution sequence can be seen in the original Petri net as the one in which the decisions at places 3 and 6 evaluate to *true* at their first encounter, whereas the decision at place 8 evaluates to *false* at first and then *true* on the second encounter. In state $s\,15$ immediately preceding the final state $s\,16$, the token in place 9 is shown to have an age of -2, indicating that for this particular minimum execution sequence, the control path through place 9 waits for 3 time units before transition 12 fires, ending the execution. The control path through place 11 does not wait because the extra time is consumed by the second execution of the loop body forming the third task.

Note that this sequence is not the only minimum execution sequence. Another one is found indirectly in the tree by going through a duplicate node. The sequence

$$s\,1, s\,2, s\,3, s\,4, s\,34, s\,35, s\,24, s\,25, s\,26$$

is also of minimum length 8. This sequence represents the execution in which all three decisions evaluate to *true* at their first encounters. Essentially, no loops bodies are executed more than once.

To illustrate the utility of the model for verifying that a software system meets specified timing constraints, let us consider the following question about the example system: how fast can the first task (coefficient acquisition, the left-hand loop in Figure 4.22) run without saturating the second task (root calculation, the center loop in Figure 4.22), thereby overwriting and losing a previous, unused set of coefficients? To answer this question, we must calculate the minimum execution time of the first task and the maximum execution time of the second to see if they possibly overlap. The coefficients are written into the shared buffer by basic block QUAD.4, represented by place p_4 in the Petri net. The minimum time between successive token arrivals at p_4 is then the minimum duration of the acquisition task. The maximum duration of task 2 cannot be greater than this or the tasks will collide. From the net structure (refer to Figure 4.22) the maximum duration of task 2 is the maximum time between successive token arrivals at place p_5, which is given by the sum of the maximum durations for p_5 and p_6. For this example, assume that the timing function τ given above is a maximum timing for the system. Then, the maximum duration of task 2 is

$$\tau(p_5)+\tau(p_6) = 5+1 = 6\,.$$

If task 1 can ever require fewer than 6 units of time to complete, then the system does not meet the requirement that no coordinate sets be lost. Given a minimum system timing τ', the minimum duration for task 1 would be

$$\tau'(p_2)+\tau'(p_3)+\tau'(p_4)\,.$$

if we take the given timing function as also representing minimum durations, then this sum is 6, indicating that the speeds of the tasks are such that no overlap will occur.

One shortcoming of these analysis techniques can be illustrated by examining the system deadlocks identified by the reachability tree. No full deadlock states exist, that is, states with a positive number of tokens but all ages non-positive. Some partial deadlocks, however, can be identified from the tree as described in a previous section. One such partial deadlock exists in the state sequence

$$s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s10, s11, s12, \cdots$$

in which control paths wait at join places 9 and 10 while the third task loops endlessly and never arrives at the join. Though this partial deadlock does indeed exist in the Petri net structure and is readily found in the reachability tree, it is an example of a sequence that could never be realized under model execution, due to data state consultation in selecting transitions to fire. As can be seen from the basic blocks in Figure 4.20, all three decisions (at places 3, 6, and 8) consult the same selector into the data state. The node selected starts with the value *false* and is later assigned *true* by task 1. The partial deadlock occurs only if the first and second tasks follow the *true* paths, but the third task continues to follow the *false* path. Given the structure of the data, this is clearly impossible. To repeat the conclusion of the deadlock detection section: all real deadlocks can be identified from the reachability tree, but some false deadlocks can be identified as well.

6.12. Summary

We describe the types of operation that are assumed to contribute to the duration of the computation represented by a node in a PFG. A consistently timed system is defined to be one in which the timings ascribed to each procedure call in a basic block agree with the figures obtained by consulting the procedure model and calculating the duration from the modeled structure. Consistency is necessary to analyze the procedure interactions in an entire system; it may not be necessary for analyzing a single procedure, depending on the information being sought.

The algorithm to generate the concurrent reachability tree introduced in the previous chapter is extended to include an age for tokens in each place. The extended tree is then used to conduct timing studies on real-time concurrent systems. Length of a path in the tree is shown to be directly proportional to the duration of the represented computation. We discuss methods of calculating from the reachability tree minimum and maximum durations for HG procedure models with various graph properties: acyclic computations; cyclic computations in which cycle traversals can be bounded; and indefinitely cyclic computations. Exact results are obtained only for acyclic computations. For cyclic computations some heuristics are described using the reachability tree and its generating Petri net structure.

The final section discusses the detection of potential deadlock, full and partial, in a procedure by searching the reachability tree of the timed Petri net. Deadlock states are identified by certain forms of marking, based on the semantics of the PFG structures that give rise to the involved places in the Petri net.

Chapter 7

CONCLUSIONS AND FUTURE DIRECTIONS

The preceding chapters present a formal model of concurrent hardware/software systems and some analysis techniques for the model. In the following sections we first summarize the main contributions of this work and the conclusions drawn from it, and then discuss several related topics that merit further study.

7.1. Summary of results

The review in Chapter 2 of existing computation models reveals several features that are desirable in a modeling system: hierarchical decomposition; separation of data, instruction segments, and control; and the passing of time as a measurable entity. A model of concurrent computation is developed in Chapter 3 which combines these features found in earlier models with some novel features. Termed the *HG model of a concurrent software system*, it is based on an extended form of Petri net, one in which times are added to the places and the execution rule allows two or more transition firings to create a single state change. The HG model has a mathematical basis in set theory and graph theory, and is intended to serve as the central formalism for a programming environment and tool set that supports development and analysis of complete real-time software systems.

A system is modeled as a collection of procedures, each of which is either a *primitive procedure*, or a *procedure model*, along with a specially designated *main procedure*. Primitive procedures have no modeled structure, but they do have a duration and a defined transformation on a data state. They represent hardware instructions in a system's host machine, or perhaps just a level in the hierarchy for which no deeper analysis is required.

Each procedure model is composed of three parts, termed *model components*. The *data model component* is a set of possible h-graphs (data states) and an initial data state. It encompasses all possible data states that can be obtained during execution. The *static program model component* is a collection of sequential blocks of procedure calls drawn from the text of a procedure, along with the data *selectors* and *procedure names* contained in those blocks. It describes the portions of an algorithm which must execute in sequence. The *control flow model component* describes the collections of blocks that can execute concurrently, and in what possible orders. It contains a timed Petri net with a concurrent transition firing rule, along with some functions to associate portions of the static program model with portions of the net.

Execution of the system consists of executing in parallel the three model components. The control flow model consults the data model for decision making at branch points, selecting a set of basic blocks from the static program model that should alter the data state. The rules of executing basic blocks are applied to the selected blocks to create a new data state from the current one. Two sequences result from execution: the sequence of control (Petri net) states, and the sequence of associated data states. Taken as a pair, a control state and a data state completely define a *system configuration*, and the two execution sequences actually define a single sequence of system configurations.

Each Petri net place in the control flow model for a procedure represents a block of contiguous procedure calls from the text of that procedure. The time τ on the place in the net represents the duration of the execution of those procedure calls, in that a token arriving at a place must reside there for τ state changes before it enables any following transitions. Allowing multiple transition firings to create a single state change, and equating a state change with one time unit, an execution sequence of a timed Petri net presents a measurable notion of computation duration.

Each procedure in an HG system model can be analyzed to the exclusion of the others in the system. In addition, the structure of the Petri net describing the control flow of a procedure allows any portion of the procedure--say, a loop body--to be extracted and analyzed as if it were a separate procedure. Inclusion of the host machine for a system in the model allows analysis of real-time computations by making available some fixed points in the hierarchy where the execution durations are known.

In Chapter 4 an algorithm expression technique is introduced for creating HG models that are well formed and easily analyzable. Termed *parallel flow graphs (PFG)*, the syntax and semantics are explained in terms of creation and deletion of different parallel paths of control. Each control path effects changes to the shared data state through the execution of basic blocks of procedure calls. Parallel flow graphs are shown to be capable of expressing several well-known parallel control structures. A translation procedure from PFGs to HG system models is also outlined. Models created from PFGs have Petri nets in the *free choice* category, a class of nets with simpler structure than the general Petri nets.

Chapter 5 contains an explanation of one analysis technique for HG system models, that of detecting and correcting conflicts among parallel basic blocks on shared structures in the data state. The detection is accomplished by the creation of a restricted form of the Petri net reachability tree, termed the *concurrent reachability tree.* The simplified approach in this chapter uses systems in which the time components of the Petri nets are assumed to be unity and are hence ignorable. The concurrent reachability tree represents only the subset of the reachability set of a Petri net which is actually obtainable under the concurrent transition firing rule. From each state in the tree, pairs of *possibly* concurrent basic blocks are identified by the marking of the places representing these blocks. We say *possibly* concurrent because the analysis is dependent only on the net structure and does not employ in any way the information contained in the data state. A control path in the Petri net may never be truly executable if the data values preclude selection of certain branches.

If a pair of blocks is found to be potentially concurrently executable, it is checked for conflicts on shared data. Two forms of conflict are considered: one block writing to a node which the other has written for later reference; and one block writing to a node which the other needs to read two or more times in succession, with no intervening writes. If conflicts are found between two blocks, then critical regions within the blocks are identified and appropriate mutual exclusion structures are automatically added to the Petri net in the control flow model to prevent the critical regions from actually executing concurrently. An extension of this procedure to more than two blocks is also discussed.

Chapter 6 concludes the development of results with a discussion of using an HG system model to answer questions about whether real-time constraints can be met by a particular hardware/software system. An extension to the construction algorithm for the concurrent reachability tree is presented which creates the tree for Petri nets with non-unity times. The extension allows for overlapping executions of basic blocks, that is, blocks which begin execution in one system state but end in some later state. The analyses of Chapter 5 apply to the tree generated in this way, since again the tree represents the obtainable states of the control flow model.

A meaning is given for the deterministic times on procedure call executions, and a method is presented for ascribing a consistent set of these times to the places in the Petri nets in a system model. With such a consistent set, algorithms are developed for determining the minimum and maximum execution durations for systems with varying degrees of structural complexity. The minimum and maximum figures can be used to determine satisfaction of time constraints for the modeled system.

7.2. For further research

The following sections briefly discuss topics for further investigation, areas opened by our work on the HG software system model. Several are concrete, experimental projects; others require additional theoretical development.

7.2.1. HG, dataflow, and other functional languages

The relationship between the imperative style of programming and functional styles needs to be examined for common properties and unifying principles. A unified model of computation (concurrent by definition, with sequential as a special case) is a desirable goal, one that would incorporate all the computation paradigms discussed in this report and provide insight into new paradigms. We hope that the work on this project can make a contribution toward achieving this larger goal.

7.2.2. Automated tools

A set of tools for software system development and analysis can be developed based on the work in this report. The modeling theory can serve as a central mechanism, common to all tools in the set. Systems may be written in any of several source languages, with tools to translate each into the HG modeling formalism. Analysis and transformation is then performed on the model itself, without regard to the original text. Finally, code can be generated directly from the verified model for different target machines.

Numerous types of tool are conceivable based on the unifying HG formalism. The translators from various source languages to the HG notation are basically compiler front ends. An analysis tool is needed to perform the conflict detection and correction described in Chapter 5. Another can perform the timing consistency checking and generation of the dual time models described in Chapter 6. Based on Wilson's work [64] a tool can be developed to advise of possible aliasing situations in a system. The generation of code from the model requires several separate tools which are essentially compiler back ends.

7.2.3. Scheduling and nondeterministic control flow models

As mentioned in Chapter 3, the HG model encapsulates the notion of scheduling in two separate decisions, made in the course of determining which configuration follows the current one during a computation. The choice technique is left unspecified, but it is required to be deterministic to match the requirement that a procedure define a function. A nondeterministic choice method could lead to several different outcomes resulting from the computation of a procedure from a single initial state, which does not denote a function. An investigation is required of the ramifications of lifting the restriction on deterministic choice, and letting a procedure represent mathematically a relation rather than a function. We conjecture that the change from a deterministic to a nondeterministic control flow model can be accomplished simply by allowing the two scheduling decisions to be nondeterminisitically performed. Thus nondeterminism can be succinctly encapsulated in the model at only two points.

Another interesting question is to discover the effect of reducing the granularity of the interleaved sequential events. Currently, the interleaving takes place on the entire procedure call level: each new system configuration is obtained from the last by combining all the concurrent basic blocks into a single, interleaved block of procedure calls. A procedure call can be viewed as non-atomic, that is, as a sequence of events like copying the argument values into the initial data state, executing the function, and then copying out the result values. The interleaving can then take place with overlapped procedure call executions. The effects of this on the data state are unclear.

7.2.4. Architecture design/description

If the hardware supporting a software system can be included in a mathematical model of the entire structure, it seems reasonable that the analysis performed on the model should suggest some hardware features that enhance the analysis and others that inhibit it. An investigation can be made into the usefulness of the HG formalism for architecture design. A machine architecture can be described as if it were a software system, with the machine instructions modeled as procedures (with the structure representing microcode) and the registers or other data structures described in the data model as h-graphs.

7.2.5. Formal language aspects of the concurrent Petri nets

Peterson [51] describes several forms of formal language associated with a Petri net structure, all assuming the single transition firing rule for execution. In its simplest form, the language of a Petri net is the set of strings formed by the transition sequences of all possible executions, where a symbol from an alphabet is associated with each transition. The concurrent firing rule employed in the HG model creates a different language due to firing *sets* of transitions. We propose to investigate the formal properties of this language. It is obvious that the concurrent Petri net language is a subset of the normal language, but the question of whether it is easier or more difficult to recognize is open.

7.2.6. Experiments with implemented systems

Much experimental work needs to be done to verify the utility of the theory described in this report. Several questions remain unanswered at this point. Since most of the Petri net analysis algorithms require time and/or space that is exponential in the size of the net, models of real systems may be too large to be practically analyzable. The hope is that by having a hierarchically decomposable formalism, and by keeping procedures small, the analyses will be tractable. Another problem is that of incorporating the communication and system synchronization overhead into the basic blocks in the model. Detailed methods for doing this need to be developed through experimentation with several different host machines.

7.2.7. Expanded timing results

Investigation of the use of the timing information in the HG system model is barely started. The work in this report is the beginning of a larger effort to automate the detection and correction of synchronization difficulties based on the overlapping in time and space of concurrent blocks. Whereas the current work treats each procedure call as having either the minimum or maximum duration for all possible calls, a more thorough treatment would attempt to employ the particular parameters in each call to get tighter bounds on its actual duration. Not only could the data state be used as an aid in this investigation, but time bounds could be added for "overhead" events like parameter transmission, which are ignored in the current work. An automated procedure might be developed, as well, for specifying and timing sections of procedures (say, loop bodies) as if they were separate entities.

Another area that was not explored in this initial investigation is the handling of faults, either in the hardware processors or in the software being modeled. In terms of a Petri net, a software fault would perhaps manifest itself as a token that "vanishes" from the net without being consumed by a transition firing, representing a request for software execution which was discarded from the system schedule for some reason. A hardware fault might be manifest as a transition which is enabled and chosen, but does not fire. Such events are not handled by existing Petri net theory. New results for such expanded execution rules must be developed before a notion of fault-tolerance can be effectively integrated with the current HG modeling theory.

7.2.8. Describe syntax/language of PFG for modeling

We propose to develop a conventional language syntax and formal semantics for PFGs, to facilitate their use as a programming notation. This skips the translation stage described in the section on tools, and encourages direct programming in the HG language. A control structure for expressing explicit nondeterminism is a possible addition to the

PFG notation described in this report. Such a feature would allow modeling of control constructs like the Ada *select* statement, and the CSP *guarded command*.

7.2.9. Extended firing rule for control flow models

The execution rule for the control flow model employed in this report requires firing as many transitions from the data-enabled set as possible at each state change. This reflects a host architecture that always has enough processing elements to satisfy all requests for block execution. A more realistic situation is to have a fixed number of processors. It is very likely that at some point in a computation more blocks will be able to execute than can be handled with the number of available processing units. To model such a system, the firing rule must be altered to allow an arbitrary subset of the data enabled transitions to fire. A mechanism must also be added to the formalism to allow the choice of such a subset to be made. The cardinality of the subset would be limited to the number of processors in the machine executing the system. We propose an investigation of the ramifications of this generalization.

7.3. Conclusion

This study of the HG model of concurrent software systems has provided an insight into the timing and verification of software systems for real-time applications. The formalism provides a vehicle for mathematically describing in a hierarchical structure the functional content of a collection of concurrent computations along with their interactions, both in shared data spaces and in overlapping execution durations. Several aspects of concurrent algorithm descriptions--data, code blocks, block relationships--are modeled as separate but interrelated components of a computation. This abstraction allows the study of properties relating to each area without the obscuring hindrance of excessive detail. For instance, the interconnection of concurrent modules can be analyzed without knowing the particular procedure calls in the modules.

Included in the model of a system is not only the software that executes but a representation of the execution-time behavior of the hardware that hosts that software. These machine-oriented portions of the model provide a low-level, accurately timed base from which an estimate of execution time bounds for the higher levels can be calculated. Though complete and exact timing results can now only be obtained for acyclic systems, the model appears to provide a formal handle on cyclic systems with which further research can make some progress.

Basing the concurrency structure of the HG theory on Petri nets provides a fairly large body of analysis techniques and results that is immediately applicable to HG systems. Generalizing the execution rule of the nets, however, provides some new areas for investigation, as in the extensions to the well-known reachability tree for representing activity durations. From the HG model of a system, several synchronization and data-sharing problems in the system can be identified and corrected. The model can then serve either as a guide for generation of correct code, or as an execution vehicle in itself. The HG system modeling theory is formal enough to be used as the unifying basis of a tool set and analysis package for a program development environment.

REFERENCES

1. *Diana Reference Manual*, Computer Science Department, Carnegie-Mellon University, and Institut Fuer Informatik II, Universitaet Karlsruhe (March 1981).

2. *Reference Manual for the Ada Programming Language*, United States Department of Defense (July 1982).

3. W. B. Ackerman and J. B. Dennis, "VAL--A Value-oriented Algorithmic Language: Preliminary Reference Manual," Technical Report MIT/LCS/TR-218, Massachusetts Institute of Technology (June 13, 1979).

4. W. B. Ackerman, "Data Flow Languages," *Computer*, **15**(2), pp. 15-25 (February 1982).

5. R. Apt, N. Francez, and W. DeRoever, "A Proof System for Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, **2**(3), pp. 359-385 (July 1980).

6. Arvind and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," pp. 849-854, in *Information Processing 77: Proceedings of the IFIP Congress 77*, ed. B. Gilchrist, North-Holland, Amsterdam (1977).

7. Arvind, K. P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," Technical Report #114A, University of California at Irvine (December 8, 1978).

8. J. Backus, "Reduction Languages and Variable Free Programming," *Report RJ 1010*, IBM Thomas J. Watson Research Center (1972).

9. J. Backus, "Programming Languages and Closed Applicative Languages," *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 71-86, ACM (1973).

10. J. Backus, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM*, **21**(8), pp. 613-641 (August 1978).

11. K. Berkling, "A Computing Machine Based on Tree Structures," *IEEE Transactions on Computers*, **C-20**(4), pp. 404 -418 (January 1971).

12. A. Bernstein, "Output Guards and Nondeterminism in 'Communicating Sequential Processes'," *ACM Transactions on Programming Languages and Systems*, **2**(2), pp. 234-238 (April 1980).

13. A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on Electronic Computers*, **EC-15**(5), pp. 757-763 (October 1966).

14. R. P. Cook, "*Mod--A Language for Distributed Programming," *IEEE Transactions on Software Engineering*, **SE-6**(6), pp. 563-571 (November 1980).

15. J. E. Coolahan, Jr. and N. Roussopoulos, "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets," *IEEE Transactions on Software Engineering*, **SE-9**(5), pp. 603-616 (September 1983).

16. T. W. Crockett and J. D. Knott, "System Software for the Finite Element Machine," NASA Technical Report CR 3870, National Aeronautics and Space Administration (February 1985).

17. J. Dennis, D. P. Misunas, and C. K. Leung, "A Highly Parallel Processor Using a Data Flow Machine Language," M.I.T. Computation Structures Group Memo TM-61 (January 1977).

18. J. B. Dennis, "Programming Generality, Parallelism and Computer Architecture," *Proceedings of the IFIP Congress 68*, pp. 484-492, IFIP (1969).

19. J. B. Dennis, "Modular Asynchronous Control Structures for a High Performance Processor," *Record of the Project MAC Conference on Concurrent and Parallel Computation*, pp. 55-80, ACM (1970).

20. J. B. Dennis, J. B. Fosseen, and J. P. Linderman, "Dataflow Schemas," pp. 187-216, in *Theoretical Programming*, Springer-Verlag, Berlin (1972).

21. J. B. Dennis, "First Version of a Data Flow Procedure Language," *Lecture Notes in Computer Science # 19: Proceedings, Colloque sur la Programmation*, pp. 362-376, Springer-Verlag (1974).

22. R. E. Filman and D. P. Friedman, *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill, New York, New York (1984).

23. D. I. Good and R. M. Cohen, "Verifiable Communications Processing in Gypsy," Certifiable Minicomputer Project Report ICSCA-CMP-11, The Institute for Computing Science and Computer Applications, the University of Texas at Austin (June 1978).

24. D. I. Good, R. W. Cohen, C. G. Hoch, L. W. Hunter, and D. F. Hare, "Report on the Language Gypsy, Version 2.0," Certifiable Minicomputer Project Report ICSCA-CMP-10, Revision 1, The Institute for Computing Science and Computer Applications, the University of Texas at Austin (September 1978).

25. D. I. Good, R. M. Cohen, and J. Keeton-Williams, "Principles of Proving Concurrent Programs in Gypsy," Certifiable Minicomputer Project Report ICSCA-CMP-15, Institute for Computing Science and Computer Applications, University of Texas at Austin (January 1979).

26. M. Hack, "Analysis of Production Schemata by Petri Nets," M.S. Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts (February 1972).

27. P. Brinch Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N.J. (1973).

28. P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, **1**(2), pp. 199-207 (June 1975).

29. P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, **21**(11), pp. 934-940 (November 1978).

30. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, **17**(10), pp. 549-557 (October 1974).

31. C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, **21**(8), pp. 666-677 (August 1978).

32. H. Jordan, "A Special Purpose Architecture for Finite Element Analysis," *Proceedings of the 1978 International Conference on Parallel Processing*, pp. 263-266 (August 1978).

33. H. Jordan, "Structuring Parallel Algorithms in an MIMD, Shared Memory Environment," Technical Report (submitted for publication), University of Colorado, Department of Computer Science (1985).

34. R. M. Karp and R. E. Miller, "Parallel Program Schemata," *Journal of Computer and System Sciences*, **3**, pp. 147-195 (May 1969).

35. R. M. Keller, G. Lindstrom, and S. Patil, "A Loosely Coupled Applicative Multiprocessing System," *Proceedings of the National Computer Conference*, pp. 861-870, AFIPS Press (1978).

36. R. M. Keller and W.-C. J. Yen, "A Graphical Approach to Software Developement Using Function Graphs," *Digest of Papers, Compcon Spring 81*, pp. 156-161 (February 1981).

37. R. B. Kieburtz and A. Silberschatz, "Comments on 'Communicating Sequential Processes'," *ACM Transactions on Programming Languages and Systems*, **1**(2), pp. 218-225 (October 1979).

38. B. Liskov, "Primitives for Distributed Computing," *Proceedings of the Seventh Symposium on Operating Systems*, pp. 33-42, Association for Computing Machinery (1979).

39. B. Liskov, "On Linguistic Support for Distributed Programs," *IEEE Transactions on Software Engineering*, **SE-8**(3), pp. 203-210 (May 1982).

40. G. A. Mago, "A Network of Microprocessors to Execute Reduction Languages, Part I," *International Journal of Information and Computer Sciences*, **8**(5), pp. 349-385 (1979).

41. G. A. Mago, "A Network of Microprocessors to Execute Reduction Languages, Part II," *International Journal of Information and Computer Sciences*, **8**(6), pp. 435-471 (1979).

42. M. Ajmone Marsan, G. Balbo, G. Conte, and F. Gregoretti, "Modeling Bus Contention and Memory Interference in a Multiprocessor System," *IEEE Transactions on Computers*, **C-32**(1), pp. 60-72 (January 1983).

43. J. R. McGraw, "The VAL Language: Description and Analysis," *ACM Transactions on Programming Languages and Systems*, **4**(1), pp. 44-82 (January 1982).

44. P. A. Merlin, "A Study of the Recoverability of Computing Systems," Technical Report 58 (Ph. D. dissertation), Department of Information and Computer Science, University of California, Irvine, California (1974).

45. R. E. Miller, "A Comparison of Some Theoretical Models of Parallel Computation," *IEEE Transactions on Computers*, **C-22**(8), pp. 710-717 (August 1973).

46. J. D. Noe and G. J. Nutt, "Macro E-Nets for Representation of Parallel Systems," *IEEE Transactions on Computers*, **C-22**(8), pp. 718-727 (August 1973).

47. J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," *Communications of the ACM*, **23**(2), pp. 92-105 (February 1980).

48. S. Owicki and D. Gries, "Axiomatic Proof Techniques for Parallel Programs," *Acta Informatica*, **6**, pp. 319-340 (June 1976).

49. S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM Transactions on Programming Languages and Systems*, **4**(3), pp. 455-495 (July 1982).

50. J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, **9**(3), pp. 223-252 (September 1977).

51. J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1981).

52. C. A. Petri, "Kommunikation mit Automaten," Ph. D. dissertation (in German), University of Bonn, Bonn, West Germany (1962).
Also, 'Communication with Automata,' C. F. Greene, Jr. (translator), Supplement 1 to Technical Report RAD C-TR-65-337, vol. 1, Rome Air Development Center, Griffiss Air Force Base, New York, N.Y., (January, 1966)

53. T. W. Pratt, "Pair Grammars, Graph Languages, and String-to-Graph Translations," *Journal of Computer and System Sciences*, pp. 560-595 (December 1971).

54. T. W. Pratt, "Application of Formal Grammars and Automata to Programming Language Definition," in *Applied Computation Theory*, ed. R. T. Yeh, Prentice-Hall (1976).

55. T. W. Pratt, "H-Graph Semantics," DAMACS Technical Reports #81-15, #81-16, University of Virginia, Charlottesville, Virginia (1981).

56. T. W. Pratt, "Formal Specification of Software Using H-Graph Semantics," pp. 314-332, in *Lecture Notes in Computer Science #153: Graph Grammars and Their Application to Computer Science*, ed. H. Ehrig, M. Nagl, and G. Rozenberg, Springer-Verlag (1983).

57. V. R. Pratt, "On the Composition of Processes," *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 213-223 (January 1982).

58. C. V. Ramamoorthy and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering*, **SE-6**(5), pp. 440-449 (September 1980).

59. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, **17**(7), pp. 365-375 (July 1974).

60. J. T. Schwartz, "Ultracomputers," *ACM Transactions on Programming Languages and Systems*, 2(4), pp. 484-521 (October 1980).

61. R. M. Shapiro and H. Saint, "A New Approach to Optimization of Sequencing Decisions," *Annual Review of Automatic Programming*, **6**(5), pp. 257-288 (1970).

62. A. Silberschatz, "Port-Directed Communication," Technical Report 50, University of Texas at Dallas (March 1979).

63. P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Surveys*, **14**(1), pp. 93-143 (March 1982).

64. J. N. Wilson, "Data Types and Aliasing in Program Specification and Verification," Ph. D. dissertation, University of Virginia, Department of Computer Science, Charlottesville, Virginia (May 1985).

65. N. Wirth, "Modula: A Language for Modular Multiprogramming," *Software—Practice and Experience*, 7(1), pp. 3-35 (January/February 1977).

66. N. Wirth, *Programming in Modula-2 (second edition)*, Springer-Verlag, Berlin (1983).

67. L. D. Wittie and A. M. van Tilborg, "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable NetworkComputer," *IEEE Transactions on Computers*, **C-29**(12), pp. 1133-1144 (December 1980).

68. W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM*, **17**(6), pp. 337-345 (June 1974).